Sather Lisp

Robert Griesemer
International Computer Science Institute, Berkeley
gri@icsi.berkeley.edu

August 17, 1994

1. Introduction

Sather Lisp is a primitive Lisp interpreter completely written in Sather 1.0. Currently no local function definitions are possible, and therefore no higher order functions can be defined directly (i.e., local lambda's do not work correctly). However, arbitrary long integers and rational numbers are fully supported.

Sather Lisp serves as an example for a non-trivial but still comprehensible Sather 1.0 application. Its simplicity lead to a small and straightforward implementation (less than 1000 lines of code, including comments and empty lines), but the interpreter is still powerful enough to give valuable insights into the principles of list processing. It can either be used to learn Sather by studying a concrete application, or to get a glance of list processing by studying the implementation details.

This documentation is only a brief description of the interpreter functionality (its user interface) and is not intended as a Lisp tutorial. The reader is referred to the standard literature for a more thourough introduction into the topic (e.g., P.H. Winston and B.K.P. Horn (1984), *Lisp*, 2nd edition, Addison-Wesley).

2. Usage

The Sather Lisp implementation consists of the file Lisp.sa and the necessary Sather library files. After compilation with:

```
cs Lisp.sa –main LISP –o Lisp <CR>
```

(assuming SATHER-COMMANDS set correctly), the Lisp interpreter can be started:

```
Lisp <CR>
Sather Lisp – gri 15 Aug 94
(symbols) returns a list of all defined symbols
>
```

The '>' prompt signals that the interpreter is ready for input. The interpreter accepts any sequence of Lisp expressions as input and evaluates them consecutively. The output is the result of the last evaluated expression (unless an error occured before). The program can be left by typing in Ctrl-D (indicating end of file) or by using the predefined function exit:

```
> (exit) <CR>
```

3. Expression syntax

The Sather Lisp syntax is described using an Extended Backus-Naur Formalism (EBNF): brackets [and] denote optionality of the enclosed sentential form, braces { and } denote its repetition (possibly zero times). Alternative forms are separated by vertical bars |. Syntactic entities (non-terminal symbols) are denoted by English words starting with a capital letter (the only one is Expression). Tokens of the language vocabulary (terminal symbols) are either denoted by English words starting with a small letter (e.g., number), or are denoted by strings enclosed in double quotes (e.g., "."). No white space may appear within tokens. However, arbitrary white space and comments may appear between tokens. Comments are denoted by the curly braces { and } and may be nested.

```
Expression  =  "'" Expression |
               "(" {Expression} ["." Expression] ")" |
               symbol | number | string.

symbol      =  letter {letter | digit} |
               special {special}.
number      =  ["–"] digit {digit} ["/" digit {digit}].
string      =  """ char """.

letter      =  "A" | "B" ... "Z" | "a" | "b" ... "z".
digit       =  "0" | "1" ... "9".
special     =  "!" | "#" | "$" | "%" | "&" | "*" | "+" | "–" | "/" | ":" |
               "<" | "=" | ">" | "?" | "@" | "\\" | "↑" | "|" | "~".
char        =  any printable character except "
```

Examples:

–1234	(negative) integer
7/31	(positive) rational number
"This is a string"	string
()	empty list, nil
myFunc	symbol
<=	(special) symbol
'a	quoted symbol
(a (b 3/4 c) d)	list consisting of 3 elements
(a . b)	dotted pair
(a b c (u v w) x . d)	list terminated with dotted pair
(a . (b . (c . ()))) = (a b c)	equivalence of dotted pair and list notation

Note: The negative sign of a number must be immediately before the first digit and must not be preceeded by another special character, otherwise it is interpreted as (special) symbol. 'x is a shortcut for (quote x) (unless quote has been redefined).

4. Expression evaluation

Sather Lisp evaluates a list by first evaluating its first element (which must evaluate to a function) and then applying the function to the remaining list. Whether the arguments are evaluated or not depends on the function; e.g., quote (') never evaluates its (single) argument, but setq evaluates only its second argument, and + evaluates all its arguments, etc. Symbols evaluate to their bound values, which may be assigned using set or setq. Initially, they evaluate to nil. Numbers, strings and functions evaluate to themselves. Truth values are denoted by the empty list (i.e., nil) and non-nil values. The empty list (nil) stands for "false", and any non-nil value stands for "true". Usually the predefined symbols nil and t (which evaluate to the empty list () and t, respectively) are used to represent "false" and "true". In order to avoid confusion, they shouldn't be redefined. If they are, (setq nil ()) and (setq t 't) re-establishes their default values.

Examples:

1234	⇒	1234	
"hello world"	⇒	"hello world"	
(+ 1 2 3 4 5 6 7 8 9 10)	⇒	55	
'(a . b)	⇒	(a . b)	
car	⇒	[car]	([...] denotes functions)
(cons 'a 'b)	⇒	(a . b)	
(car (cons 'a 'b))	⇒	a	
(cdr (cons 'a 'b))	⇒	b	
(cdr '(a b))	⇒	(b)	
(setq a 4/6)	⇒	2/3	
a	⇒	2/3	
(lambda (x) (+ x 1))	⇒	[((+ #0 1))]	(#i denotes argument no. i)
((lambda (x) (+ x 1)) 1)	⇒	2	
(setq inc (lambda (x) (+ x 1)))	⇒	[inc]	
(inc 1)	⇒	2	

5. Predefined functions

Several functions are predefined in Sather Lisp. Since predefined functions are bound to ordinary symbols, (the values of) these symbols may be redefined if desired (and the old function is lost unless it is bound to another symbol, too). User defined functions may be added (or redefined, resp.) by binding lambda expressions to symbols (see Section 6). A list of all symbols known to the system is obtained using (symbols). The following table gives a brief definition of all predefined functions.

Expression	*Result*	*Constraints*
(+ arg_0 arg_1 [... arg_n])	(arg_0 + arg_1) + ... + arg_n	(numbers only)
(– arg_0 arg_1 [... arg_n])	(arg_0 – arg_1) – ... – arg_n	(numbers only)
(∗ arg_0 arg_1 [... arg_n])	(arg_0 ∗ arg_1) ∗ ... ∗ arg_n	(numbers only)
(/ arg_0 arg_1 [... arg_n])	(arg_0 / arg_1) / ... / arg_n	(numbers only)

(% arg_0 arg_1 [... arg_n])	(arg_0 % arg_1) % ... % arg_n	(numbers only)
(↑ arg_0 arg_1 [... arg_n])	(arg_0 ↑ arg_1) ↑ ... ↑ arg_n	(numbers only)
(= arg_0 arg_1 [... arg_n])	arg_0 = arg_1 = ... = arg_n	(numbers and strings only)
(# arg_0 arg_1 [... arg_n])	arg_0 # arg_1 # ... # arg_n	(numbers and strings only)
(< arg_0 arg_1 [... arg_n])	arg_0 < arg_1 < ... < arg_n	(numbers and strings only)
(<= arg_0 arg_1 [... arg_n])	arg_0 <= arg_1 <= ... <= arg_n	(numbers and strings only)
(> arg_0 arg_1 [... arg_n])	arg_0 > arg_1 > ... > arg_n	(numbers and strings only)
(>= arg_0 arg_1 [... arg_n])	arg_0 >= arg_1 >= ... >= arg_n	(numbers and strings only)
(! arg)	factorial of floor(arg)	(numbers only)
(floor arg)	floor of arg	(numbers only)
(ceiling arg)	ceiling of arg	(numbers only)
(car arg)	head of arg	(arg must be a list)
(cdr arg)	tail of arg	(arg must be a list)
(cons arg_0 arg_1)	(arg_0 . arg_1)	
(atom arg)	t if arg is not a pair, () otherwise	
(eq arg_0 arg_1)	arg_0 = arg_1	
'arg	shortcut for (quote arg)	
(quote arg)	returns arg without evaluation	
(eval arg)	evaluates the *value* of arg	
(set sym arg)	binds the value of arg to the symbol sym	
(setq arg_0 arg_1)	binds the value of arg_1 to the *symbol value* of arg_1	
(write arg)	writes arg to stdout and returns arg	
(writeLn)	starts a new line to stdout and returns ()	
(lambda ...)	see Section 6	
(cond arg_0 [arg_1 ... arg_n])	each argument is considered to be a list consisting of a condition (which evaluates to nil or non-nil) and a sequence of expressions. cond evaluates the first sequence of expressions for which its condition is evaluated to a non-nil value. The result of cond is the last expression evaluated (see examples).	
(readFile arg)	reads and evaluates file arg	(arg must be a string)
(tracer arg)	arg = on: turns tracing output on and returns on arg # on: turns tracing output off and returns off	
(tracer)	actual tracing mode (on or off)	
(symbols)	returns list of all known symbols	
(exit)	exits the interpreter	

Note: Use = for comparison of numbers and strings; eq does only a pointer comparison, thus (eq 1 1) # t!

6. Function definition

New functions can be created using the *lambda* function. lambda expects a parameter list and a sequence of expressions. Three syntactic variants exist:

a) (lambda x $expr_0$ $expr_1$ ... $expr_m$)
b) (lambda (par_0 par_1 .. par_{n-1}) $expr_0$ $expr_1$ ... $expr_m$)
c) (lambda (par_0 par_1 .. par_{n-1} . x) $expr_0$ $expr_1$ ... $expr_m$)

a) This corresponds to case c) with n = 0.
b) The function expects n parameters which are hold in par_0 to par_{n-1}. The result of a function invocation is the value of the last expression $expr_m$ evaluted using the parameter *values* (i.e., the parameters are evaluated before the $expr_i$'s are evaluated). Symbols that are not parameters are considered to be global.
c) The function expects at least n parameters. The *values* of the first n parameters are hold in par_0 to par_{n-1} (i.e., these parameters are evaluated before the $expr_i$'s are evaluated), the remaining *parameter list* is hold in x (i.e., the last parameter is *not evaluated* prior to evaluation of the $expr_i$'s).

Variant a) and c) can be used to implement functions that accept variable long argument lists or functions that do only partially evaluate their arguments. For instance, quote could be defined by (setq quote (lambda x (car x))). The ReadFiles function definition in the example section below is another application of this feature. Furthermore, variant a) and c) can also be (mis-)used to implement functions with one local variable x. If more than one local variable are required, an auxiliary function must be called that specifies local variables as additional arguments which are not used (but initialized) by the call.

Note: A function which is used several times should be bound to a symbol (for efficiency reasons). This is especially important when using recursion.

Implementation restriction: Currently, local lambda definitions (lambdas within lambdas) do not work correctly. Furthermore, quoting arguments within lambdas is erroneous. Both these problems are related and due to the simplified implementation. Solving these problems correctly requires a redesign of the function evaluation mechanism.

Examples:

```
(lambda (x) (add x 1))                  defines an increment function
(setq inc (lambda (x) (add x 1)))       the value of inc is the increment function
(inc 1)                                 inc applied to 1 returns the value 2

(setq sum                               defines a function sum which recursively
  (lambda (n)                           calculates the sum of the first n integers
    (cond
      ((< 0 n) (+ n (sum (- n 1))))
      (t 0)
    )
  )
```

```
)

(sum 100)                          5050

(setq list (lambda x x))           returns its arguments as a list without evaluation
(list a b c)                       (a b c)

(setq readFiles                    extend the readFile function to arbitrary many
  (lambda x                        arguments
    (cond
      ((atom x))
      (t (readFile (car x)) (eval (cons 'readFiles (cdr x))))
    )
  )
)
```

7. Error recovery

Currently, after a parse- or run-time, the interpreter only prints out a short error message and returns to the read-eval-write loop. If the error hapens during file input (i.e., during the evaluation of a (readFile arg) expression, also the file name (arg) is displayed. However, the Lisp tracing facility can be used to simplify the location of bugs. The function tracer is turned on by evaluating (tracer on), and turned off by evaluating (tracer off). When on, the interpreter prints out the every (predefined and user-defined) function call together with its arguments and the return value. The following shows the trace of the expression (sum 3) (sum is defined in Section 8):

```
> (tracer on)
on
> (sum 3)
[sum] called with (3)
  [cond] called with (((<= #0 0) 0) (t (+ #0 (sum (– #0 1)))))
    [<=] called with (#0 0)
    [<=] returns nil
    [+] called with (#0 (sum (– #0 1)))
      [sum] called with ((– #0 1))
        [–] called with (#0 1)
        [–] returns 2
        [cond] called with (((<= #0 0) 0) (t (+ #0 (sum (– #0 1)))))
          [<=] called with (#0 0)
          [<=] returns nil
          [+] called with (#0 (sum (– #0 1)))
            [sum] called with ((– #0 1))
              [–] called with (#0 1)
              [–] returns 1
              [cond] called with (((<= #0 0) 0) (t (+ #0 (sum (– #0 1)))))
                [<=] called with (#0 0)
```

```
                [<=] returns nil
                [+] called with (#0 (sum (– #0 1)))
                  [sum] called with ((– #0 1))
                    [–] called with (#0 1)
                    [–] returns 0
                    [cond] called with (((<= #0 0) 0) (t (+ #0 (sum (– #0 1)))))
                      [<=] called with (#0 0)
                      [<=] returns t
                    [cond] returns 0
                  [sum] returns 0
                [+] returns 1
              [cond] returns 1
            [sum] returns 1
          [+] returns 3
        [cond] returns 3
      [sum] returns 3
    [+] returns 6
  [cond] returns 6
[sum] returns 6
6
>
```

Function names (i.e., the symbols to which the functions are bound to) are shown in brackets []. If there is no name for a particular function (i.e., in case of a anonymous lambda expression), the function definition is printed out instead. Parameters are denoted by #i, starting with i = 0 for the first parameter of a lambda expression. If the same function sum is called with a wrong argument, e.g., a string, the run-time error can be located easily:

```
> (sum "illegal argument")
[sum] called with ("illegal argument")
  [cond] called with (((<= #0 0) 0) (t (+ #0 (sum (– #0 1)))))
    [<=] called with (#0 0)
error in [sum]: 0 is not a string
>
```

Obviously, the error occured in the user-defined function sum (error messages refer to user-defined function names only). Within sum, the last function called was [<=] with the arguments #0 and 0. Since #0 refers to the string "illegal argument", the expression to be evaluated is (<= "illegal argument" 0) which results in a run-time error, since [<=] requires all arguments to be either numbers or strings.

8. A few more examples

{ *Sum of the first n integers* }

```
(setq sum
  (lambda (n)
```

```
      (cond
        ((<= n 0) 0)
        (t (+ n (sum (– n 1))))
      )
    )
  )

{ Examples }

(sum 0)
(sum 10)
(sum 100)
(sum 1000)
(sum 0)


{ Factorial using Peano Axioms }

(setq s
  (lambda (x)
    (cons 's (cons x nil))))

(setq p
  (lambda (x)
    (car (cdr x))))

(setq myAdd
  (lambda (x y)
    (cond
      ((atom x) y)
      (t (s (myAdd (p x) y))))))

(setq myMul
  (lambda (x y)
    (cond
      ((atom x) 0)
      (t (myAdd (myMul (p x) y) y)))))

(setq gen
  (lambda (n)
    (cond
      ((<= n 0) 0)
      (t (s (gen (– n 1)))))))

(setq fact
  (lambda (x)
    (cond
      ((atom x) (s 0))
```

```
      (t (myMul x (fact (p x)))))))
```

{ *Examples (gen is used to create Peano Integers)* }

```
(gen 0)
(gen 2)
(gen 10)
(gen 100)

(p (gen 3))
```

{ *Peano addition* }

```
(myAdd (gen 3) (gen 4))
(myAdd (gen 0) (gen 2))
```

{ *Peano multiplication* }

```
(myMul (gen 2) (gen 3))
```

{ *Peano factorial* }

```
(fact (gen 0))
(fact (gen 2))
(fact (gen 4))
(fact (gen 5))
(fact (gen 6))
```

{ *Towers of Hanoi* }

```
(setq print
  (lambda (x)
    (write x) (writeLn)))

(setq move
  (lambda (from to)
    (print (cons from (cons to ())))))


(setq hanoi
  (lambda (from over to n)
    (cond
      ((> n 0)
        (hanoi from to over (- n 1))
        (move from to)
        (hanoi over from to (- n 1))
      ))))
```

{ *Example* }

```
(hanoi 'a 'b 'c 5)
```

{ *Ackerman function* }

```
(setq A
  (lambda (x y)
    (cond
      ((= x 0) (+ y 1))
      ((= y 0) (A (– x 1) 1))
      (t (A (– x 1) (A x (– y 1))))
    )
  )
)
```

{ *Examples* }

```
(A 3 2)
(A 3 3)
```