Superscript Language Reference Manual

Uday Singh | Samurdha Jayasinghe | Tommy Orok | Michelle Zheng | Yu Wang
(urs2102) | (sj2564) | (to2240) | (myz2103) | (yw2684)

1. Introduction

Superscript is a type-inferred Lisp with syntax, focused on rapid development, that compiles into Javascript. Superscript is a compiled language with static type inference, using the Lisp core, that allows the user to write robust, yet concise programs. It is heavily influenced by both Arc and Clojure, two new languages introduced to the Lisp community, and OCaml.

Unlike Javascript, the functional-first nature of Superscript encourages users to think more before they code. Using static type inference based on the Hindley-Milner algorithm, our compiler tells users when their functions break due to type inconsistency. Similar to OCaml, it discourages users from thinking in terms of objects, and encourages clean data transformations through the use of S-expressions, which may be written using either Lisp-like, or imperative syntax (refer to Section 8, for Syntax Modifications to the Lisp core). This, combined with the flexibility of the Lisp family of languages, where functions and data are equivalent, gives users of Superscript a level of power unavailable in languages like Javascript.

Superscript enables sharp programmers to think and express clear functional ideas in a language that is slowly becoming the backbone of the web.

Program Structure

Superscript programs are built with expressions, and a program can be viewed as a single expression or a list of expressions executed in sequence. In Superscript, functions are data and data are functions.

In the following examples, the indicated result is what the expression would evaluate to, in the executable JavaScript produced by the Superscript compiler.

```
; Function call that prints Hello, world!
> (prn “Hello, world!”)
Hello, world!

; Addition function applied to all integers from 1 to 10
> (+ 1 2 3 4 5 6 7 8 9 10)
55

; Function call, applying the ‘head’ function to a list
```

```
; that consists of ‘+’ and the integers 1 through 10.
> (head ‘(+ 1 2 3 4 5 6 7 8 9 10))
+
```

The idea that both functions and data are one and the same allows for rapid expression of ideas in Superscript that would otherwise be more difficult to express in Javascript. The above examples show not only how functions are executed, but how comments work (using the semicolon), and the power of code and data being one.

2. Lexical Conventions

2.1 Reserved Keywords

Superscript has a set of reserved keywords and function names that cannot be used as symbols (identifiers for variables or functions).

Reserved Keywords	Standard Library Functions	Primary Lisp Functions
true, false, nil	fn, def, if, while, loop, let, with, and, or, not, isnt, iso, mod, prn, do, repeat, type	quote, atom, is, head, tail, cons, if

2.2 Punctuation

2.2.1 Semicolon

Comments in Superscript start with a semicolon and terminate with a new line:
>1 ; This is a comment
1

2.2.2 Double semicolon

A program in Superscript is a list of expressions, and the expressions are separated by double semicolons:
> (prn “hello”);; (prn “world”);; (prn “people”);;
hello
world
people

2.2.3 Parentheses

Parentheses must be part of the representation of a list, which must be quoted if it is not intended to be a function call:
> ‘(1 2 3)
(1 2 3)

Parentheses must be used to wrap a function definition, to wrap its arguments, and to wrap the function body:
> (fn (x y) (+ x y))

Parentheses must be used to wrap any function call, including a standard library function call or primary Lisp function call:
> (average 2 4) ; average is a function that returns the average value of its parameters
3

> (if true 1)
1

All unquoted lists are interpreted as function calls. Hence, parentheses cannot be used to wrap an unquoted list that is not a function call. The following is a syntax error, and cannot be evaluated:
> (1 2 3)

2.2.4 Curly Braces

Braces can be used to wrap an infix expression and to explicitly indicate the order of operations and associativity. See Section 8 for more information about syntactic sugar and Syntax Modifications that allow a more imperative programming syntax.

> {{1 + 2} * {3 - 2} * 8}
24

3. Data Types

3.1. Type Inference

Superscript uses static type inference, based on the Hindley-Milner algorithm[12], to evaluate the type and value of any legal expression and to check that expressions satisfy the proper data type in all function calls. Our compiler will flag any inconsistent data type in function calls as error. Based on this type inference, the Superscript compiler implements error handling and prints the appropriate error messages to the user.

Superscript's standard library provides a set of functions to let user convert between different data types.

3.2. Atomic Data Types

The following are atomic (non-list) data types in Superscript.

3.2.1 Numerical types

We use type inference to avoid NaN errors common in Javascript, since non-numerical values cannot be used in arithmetic functions.

Integers
Superscript allows integers, which may be manipulated by integer arithmetic using the standard +, -, /, * operators. Integers are sequences of digits containing no decimals. Integers are considered accurate up to 15 digits.

```
> 42
42

> (type 42)
‘int’
```

Floating Points
Superscript allows floating point values and requires the use of floating point specific operators to perform arithmetic. These operators are the int operators followed by ‘.’ (+., -., /., *.). Floating points must include at least one digit and a decimal, satisfying the regular expression:

['0'-'9']*'.'['0'-'9']+ | ['0'-'9']+'.'['0'-'9']*

```
> 42.0
42.0

> (type 42.0)
‘float’
```

3.2.2. Integer and Floating Conversions

When a value of floating type is converted to integral type, the fractional part is discarded. When a value of integral type is converted to floating, if the value is not exactly representable, the result is the next higher or next lower representable value. Using type inference, if either operand of an arithmetic expression is float, then the other is converted to float. For instance, int + float will be converted to float +. float. If two operands are both integral, but the floating point

operator is used, then floating point arithmetic is performed: int +. int becomes float +. float. If both operands are int, then integral arithmetic is performed.

3.2.3. String

Superscript supports ASCII strings as a way to represent textual data. Like Javascript, a single character is treated as a single character String. There is no type for chars. Print always prints to console and returns the nil datatype.

```
> “Hello, world!”
Hello, world!

> (type “Hello, world!”)
‘string’
```

3.2.4. Symbols

Symbols, or identifiers, represent programmer-defined objects.. They are containers for storing data values, and they return the value assigned to them. An identifier name must start with a letter, followed by any number of letters, digits, underscores or hyphens, is defined by the regular expression:

['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_' '-']*

```
> (= a 42)
42

> a
42
```

Evaluation for symbols can be turned off by putting a single quote character (‘) before an expression.

```
> ‘a
“a”
```

3.2.5. True/False/Nil

Superscript has a boolean true and false value.

```
> (type true)
‘boolean’
```

```
> (type false)
‘boolean’
```

Superscript also has a nil value which is the null datatype, returned by certain functions such as print. It is equivalent to quoting the empty list.

```
> nil
nil

> ‘()
nil
```

3.3. Non-atomic Data Types

3.3.1. Lists

Multiple atoms enclosed parentheses are also called lists. An unquoted list is a function call, where the first element is the function name, and the other values are the parameters (See 3.3.2). For instance, *(a b c d e)* calls the function ***a*** with the arguments ***b***, ***c***, ***d***, and ***e***. It is a syntax error to write an expression that is an unquoted list, such as (5 4 8), where the first element is not a function name.

In order to use (a b c d e) as a list, rather than function call, you must quote the list: ‘(a b c d e) is a list of the elements a, b, c, d, and e.

```
> ‘(1 2 3)
(1 2 3)

> (type ‘(1 2 3))
‘list’
```

3.3.2. Functions

Call a function using an unquoted list, where the first argument is the function name, the following arguments are the parameters passed into the function call, and all the parameters are passed in by value, in the format of (function_name arg1 arg2 arg3…):

```
> (sum 1 2 3 4)
10
```

Define an anonymous function in the following way, using the ‘fn’ keyword: (fn (optional_args) (expression)), where optional_args is 0 or more formal arguments separated by spaces, and enclosed by parentheses; the body of the function is a single expression enclosed by

parentheses; and the entire expression is surrounded by parentheses. A function definition returns a function data type. In Superscript, functions are a data type much like lists and atoms.

```
> (fn (x y) (/(+ x y) 2))
[Function]
```

We can bind an anonymous function declaration to a name using the '=' function, which evaluates right-to-left. Don't be scared by the prefix notation, as the same may be expressed using infix notation, covered in Section 8, Syntax Modifications.

Anonymous function declaration, which is then bound to the name 'average':

```
> (= average (fn (x y) (/ (+ x y) 2)))
[Function]
```

Function call:

```
> (average 20 10)
15
```

You can also use the shorthand *def* to define named functions. *def* takes the function name as the first argument, a list of arguments as the second argument, and the function body as the third argument. *def* is just syntactic sugar for assigning an anonymous function to the symbol 'average'.

```
> (def average (x y)
      (/ (+ x y) 2))

> (average 20 10)
15
```

3.3.3. Tables

Although lists can be used to symbolize most data structures, the most efficient way to store a key/value pair is through a hash table. They are also useful for building objects in Javascript; however, in Superscript, it is better to view them as a data type for key/value pairs.

```
> (table)
{}

> (= airports (table))
{}

> (= (airports "New York") jfk)
'jfk'
```

Use the ‘obj’ function to define a table with greater ease. This will return the last element.

```
> (= codes (obj "Boston" 'bos "San Francisco" 'sfo "New York" 'jfk))
jfk

> (codes)
{ ‘Boston’: 'bos', ‘San Francisco’: ‘sfo’, New York': 'jfk' }

To get the keys
> (keys codes)
[ ‘Boston’, ‘San Francisco’, ‘New York’ ]

To get a value
> (codes ‘Boston’)
‘bos’
```

4. Operators and Built-in Functions

4.1. Basic Assignment

This is a basic assignment that sets the value of ***a*** to equal ***b***’s **current value**.

```
> (= a b)
b

> (= x 5)
5
```

Basic assignment is applicable to all datatypes.

4.2. Arithmetic Operators

Before we go into arithmetic operators, note that we will be using prefix notation here. Superscript allows infix notation as well under Section 8.

Superscript offers several Standard Library arithmetic functions. These include both integral and floating addition, subtraction, multiplication, division; as well as the modulo of two ints. These functions are used as follows.

Addition, Subtraction, Multiplication, Division

You can add at least 2 arguments, ***a*** to ***b***, or add an unlimited amount of arguments together. The following examples show function calls to arithmetic operations.

(+ a b)

(+. a b c d e f g h i j k l m n o p …)

Both of the above expressions are unquoted lists where the first element is a function call to the addition function, done on the other elements in the list. The first example uses the integer operator (+), and the second, the floating operator (+.). Addition, Subtraction, Multiplication, and Division can be applied to two or more arguments.

All arguments to arithmetic functions must be numerical. They will be evaluated left to right. For the rules of integral and floating conversion, see Section 3.2.2.

Modulo

You can call a modulus function between two integers ***a*** and ***b,*** this will return the remainder of a / b.

```
> (mod 5 6)
5
```

4.3. Boolean Operators

Superscript's Standard Library contains logical functions to evaluate boolean expressions.

4.3.1. Logical NOT Function

'not' is a Standard-Library function that takes one boolean expression as its argument, and negates the value of that expression.

```
> (not true)
false

> (not false)
true
```

4.3.2 Logical AND / OR

Superscript's Standard Library supports logical AND/OR functions, using 'and' and 'or' keywords.

```
> (and true false)
false
```

```
(or true true)
> true
```

4.3.3 Equality and Inequality

The ‘is’ function is an equality comparison that may be applied on atomic constants: ints, floats, and strings. The ‘isnt’ function compares ints, floats, and strings for inequality.

```
> (is “a” “b”)
false

> (is 1 1)
true

> (is 1 2)
false

> (isnt “a” “b”)
true
```

4.3.4 Isomorphic List Equality

The isomorphic list equality function, which must be called on lists, will return true if all the elements for the each of lists are equal.

```
> (iso (list ‘a) (list ‘a))
true
```

4.3.5 Relational Comparison operators

These relational comparison operators can be used for ints, floats, and strings. They take two expressions as arguments:

```
(> a b) (< a b) (>= a b) (<= a b)

> (> 5 4)
true
```

4.4 String concatenation

String concatenation is done using the “++”’ function.

```
> (++ “hello” “ ” “world”)
```

```
“hello world”
```

4.5. prn/pr Function

Print always prints to console. Its return type is nil.

```
> (prn “Hello, world!”)
Hello, world!

> (prn 5)
5

> (type (prn 5))
nil
```

4.6 Infix Operations

See Section 8, Syntax Modifications, for ways to use syntactic sugar to write infix expressions for all of the above operators, along with their precedence and associativity in infix notation.

5. Primary Lisp Functions

5.1. quote Function

`(quote a)` returns **a**. For shorthand, abbreviate it as **‘a**. You must quote values if you want the S-expression not to be evaluated, but returned instead.

```
> (= a 42)
42

> (quote a)
a

> (a)
42

> (‘a)
a
```

5.2. atom Function

(atom ‘a) returns the atom true if the type of **a** is an atom. This function will also return true when applied to the empty list.

```
> (atom ‘a)
true
> (atom ‘(a b c))
false
> (atom ‘())
true
```

5.3. is Function

`(is a b)` returns true if the value of **a** equals that of **b**. Returns false otherwise.

```
> (is ‘a ‘a)
true
> (is ‘a ‘b)
false
```

5.4. head Function

(head a) expects **a** to be a list, and returns its first element.

```
> (head ‘(a b c))
a
```

5.5. tail Function

(tail a) expects **a** to be a list of n values and returns a new list comprised of the 2nd to the nth elements of **a**.

```
> (tail ‘(a b c))
(b c)
```

5.6. cons Function

(cons a b) expects the value of **b** to be a list, and returns a list comprised of **a,** as the first element, followed by the elements of list **b.**

```
> (cons ‘a ‘(b c))
(a b c)

> (cons ‘(a) ‘(b c))
((a) b c)
```

5.7. if Function

(if a b c) where **a** is an S-expression which returns a boolean; equivalent to “if **a** then **b** else **c**”.

```
> (if (odd 1) ‘a ‘b)
a
```

(if a b c d e) where **a** is an expression and **b** is a return value considers **c** to fall as a condition for an else-if statement with the result of **c** being true resulting in **d** and with the else of the entire if statement being **e.**

`(if a b c d e)` is equivalent to:

```
(if a
    b
    (if c
        d
        e))
```

5.8. do Function

(do a b c ...) is a function which runs the argument functions sequentially. It runs, **a**, then **b**, then **c** sequentially. It returns the last value. Although I have used only three arguments here, ‘do’ can take unlimited arguments.

```
> (do (prn "where") (prn "are") (prn "the") prn(“spaces”))
wherearethespaces
```

5.9. Temporary Variables Function

There are two commonly used functions for establishing temporary variables, let and with. These temporary variables’ usage is only limited to some specific scope as described below.

5.9.1 let Function

(let a b c) is a function where the value of **b** is assigned to variable **a** to be used temporarily in the function **c**. The value of a becomes **Nil** outside of this expression.

```
> (let x 1 (+ x (* x 2)))
3
```

5.9.2 with Function

(with a b) is a function where **a** is a list of symbols and expressions whose length must be even, and that assigns the value of expression at each even position to the symbol before it and creates a list of temporary values which are used in the function **b**.

```
; Assuming that sqrt is a square root function and expt is an
; exponentiation function
> (with (x 3 y 4) (sqrt (+ (expt x 2) (expt y 2))))
5
```

6. Iteration

6.1 while Function

(while predicate body) is a function that repeatedly executes the body, as long as the predicate is true. The predicate is evaluated before each iteration of the body.

```
> (let x 10
      (while (> x 5)
            (= x (- x 1))
            (pr x)))
98765
```

6.2 loop Function

(loop initialization condition update body) is a function that repeatedly loops on the body until a particular condition is satisfied. It first executes the initialization, once, as the loop begins. When the termination condition evaluates to false, the loop terminates. The update expression is executed after each loop iteration.

```
> (loop (= x 0) (< x 3) (= x (+ x 1)) (pr x))
012
```

6.3 repeat Function

(repeat count body) is a function where the body is executed count number of times.

```
> (repeat 3 (pr “lol”))
lollollol

> (do (repeat 8 (pr “Na”)) (pr “ Batman!”))
NaNaNaNaNaNaNaNa Batman!
```

7. List Operations

7.1. Head/Tail Functions

Refer to 5.4 and 5.5, on Lisp Functions.

```
> (List.head `(a b c))
a

> (List.tail `(a b c))
(b c)
```

7.2 Getting element at index

(List.get a b) The get function allows you to retrieve the **a**'th value from the list **b**.

```
> (List.get 0 ‘(a b c d))
a

> (List.get 3 ‘(a b c d))
d
```

7.3. Joining Lists

(List.join a b c …) is a function which joins the list values in **a**, **b**, **c**, and more, to form a list by appending them from left to right.

```
> (List.join ‘(1 2) ‘(3 4))
(1 2 3 4)
```

7.4 Reduce Function

(List.reduce a b) is a function which reduces the list **b** using the function **a**. It applies the function **a** to the first two elements of **b** and recursively applies **a** to that result and the next element in **b**.

```
> (List.reduce + ‘(1 2 3 4 5))
15

> (List.reduce /. ‘(1.0 2.0 3.0))
0.16666666666666666
```

7.5 Right-reduce Function

(List.rreduce a b) is a function which reduces the list **b** using the function **a**. It applies the function **a** to the last two elements of **b** and recursively applies **a** to that result and the previous element in **b**.

```
> (List.rreduce + ‘(1 2 3 4 5))
15

> (List.rreduce /. ‘(1.0 2.0 3.0))
1.5
```

8. Caramel, or “Syntax Modifications”

“It’s hot syntactic sugar for Superscript ” - Author Unknown

Throughout this manual, we have introduced you to the basics of Superscript. Superscript is obviously a Lisp, and for that reason programs in Superscript are composed of S-expressions. We have an example of a function call written below.

```
> (operator argument argument argument)
```

A list of the same values looks like the following:

```
> ‘(operator argument argument argument)
```

By simply quoting the first list, we do not look at it as a function call, but a list. This type of syntax, we believe, is both one of Superscript’s greatest strengths but also one of its greatest shortcomings. Making programs look like data structures is something that is extremely attractive to not just the Lisp community, but to someone who understands how our language’s programs are almost prewritten in their AST.

The shortcoming of this type of syntax is that without either the ability to separate parentheses or the user understanding the idea of constantly applying a function to a list of arguments, things like basic arithmetic seem less natural.

With Superscript attempting to be a language which focuses on simplicity allowing users to “do a lot with little”, Superscript has an alternate syntax mode which prevents ‘parenthetical hell’ without losing the capabilities of Lisp.

Superscript already utilizes abbreviations to common Lisp ideas, such as using ‘x instead of (quote x), so additional abbreviations have been added so that programs in Superscript using Caramel syntax can be efficiently parsed without losing the semantic power of Lisp.

You can use any of these features to make your code easier to read and write. This reduces the amount of tokens necessary to program thus following Superscript’s key principle of reducing the number of tokens needed to express an idea.

We will start with a program written in S-expressions and then break it down using Caramel to demonstrate the power of our extremely sugared syntax system.

There are three key components which can be layered on each other and used with or without each other.

1. **Curly infix expressions (c-expressions)**
2. **Modern Expressions (m-expressions)**
3. **Indented Expressions (i-expressions)**

```
; Let's start with this function
(def fac (n) (if (<= n 1) 1 (* n (fac (- n 1)))))
```

8.1. Curly infix expressions

Rule: Arithmetic infix operations can be put in a single curly expression or comparator operations can be put in a single curly expression.

a. If arithmetic operations are used they are evaluated in order of operations.
b. If comparator operations are used, there may only be two elements for the function to be evaluated.
c. If you attempt to mix comparators and arithmetic operations, you will receive an error.

```
> {1 + 2}
3

> {1 < 2}
true
```

The factorial function defined as follows:

```
(def fac (n)
      (if (<= n 1)
            1
            (* n (fac (- n 1)))))
```

Can be made more readable through the use of {} expressions:

```
(def fac (n)
      (if {n <= 1}
            1
            {n * (fac {n - 1})}))
```

8.2. Modern Expressions

Modern expressions can be built using the following rule:

Rule: Anything outside a bracket without a space gets slurped into an s-expression.

a. f(…) -> (f …)

b. f{…} -> (f {… }) where {…} sees rule 1.

The factorial function thus far:

```
(def fac (n)
      (if {n <= 1}
            1
            {n * (fac {n - 1})}))
```

Can be evaluated to:

```
def (fac (n)
      if {n <= 1}
            1
            {n * fac ({n - 1})})
```

8.3. Indented Expressions

Rule: Parentheses are derived by indentation. All indented lines are parameters of parent lines. To understand Parent and child lines see this:

a. If terms are on the same line, they are parameters of the first term.

b. If a line has one term and no other operators, it returns itself.

c. To execute the final greater expression leave an empty line or a double semi-colon.

```
parent_line
      child_line
            grand_child_line
```

Take this:

```
def (fac (n)
      if {n <= 1}
            1
            {n * fac ({n - 1})})
```

And do this:

```
def fac(n)
   if {n <= 1}
     1
     {n * fac{n - 1}}
```

Final example:

```
; old way
(def fac (n) (if (<= n 1) 1 (* n (fac (- n 1)))))

; new way
def fac(n)
   if {n <= 1}
     1
     {n * fac{n - 1}}
```

8.2 Operator Precedence and Associativity

The table below shows all Superscript operators, from lowest to highest precedence, along with their associativity.

Operator	**Description**	**Associativity**
=	basic assignment	right
or	logical OR	left
and	logical AND	left
is isnt	equality comparison inequality comparison	left
> < >= <=	relational comparison	left
++	string concatenation	left
+ - +. -. mod	integer addition, subtraction float addition, subtraction modulo	left
* / *. /.	integer multiplication, division float multiplication, division	left
not	logical NOT	right

Footnotes & References

[0]: Javascript is Assembly Langauge for the Web http://www.hanselman.com/blog/JavaScriptIsAssemblyLanguageForTheWebPart2MadnessOrJustInsanity.aspx

[1]: 7 Principles of Rich Web Applications. Guillermo Rauch. http://rauchg.com/2014/7-principles-of-rich-web-applications

[2]: Wat, a lightning talk by Gary Bernhardt. https://www.destroyallsoftware.com/talks/wat

[3]: The Roots of Lisp. Paul Graham. http://languagelog.ldc.upenn.edu/myl/ldc/llog/jmc.pdf

[4]: Arc. Paul Graham. http://www.paulgraham.com/arc.html

[5]: Insertion Sort Lisp Implementation. Bob Dondero. http://cs.princeton.edu/courses/archive/spr11/cos333/lectures/17paradigms/sort.lisp

[6]: Clojure for the Brave and True. Daniel Higginbotham http://www.braveclojure.com/

[7]: Arc Language. http://arclanguage.github.io/

[8]: Arc Forum. http://arclanguage.org/forum

[9]: Curly infix, Modern-expressions, and Sweet-expressions: A suite of readable formats for Lisp-like languages, David A. Wheeler: http://www.dwheeler.com/readable/sweet-expressions.html

[10]: History of Lisp. John McCarthy. http://www-formal.stanford.edu/jmc/history/lisp/lisp.html

[11]: Memo 8: Recursive Functions of Symbolic Expressions and Their Computation By Machine, John McCarthy. ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-008.pdf

[12]: Hindley Milner type inference. http://c2.com/cgi/wiki?HindleyMilnerTypeInference