Setting Up To Build Prompts

Scheme : list based programming language, focus on recursion

Basic Data Types :

1. atom : string of characters, letters, digits

* eg : turkey, 1492, \*abc$

1. list : collection of S-expressions enclosed by parentheses

* eg : (word is not), ((word) is (not))
* () is the null list

1. S - expressions : an atom or list is a S - expression

Primitives :

1. car : gives first S-expression of a list

* defined only for non - empty lists, returns a S - expression
* eg : car of l where l is (((hotdogs)) (and) (pickle) relish) = ((hotdogs))

( car ( car l)) where l is (((hotdogs)) (and)) = (hotdogs)

1. cdr : gives list l without (removing) car l

* defined only for non - empty lists, returns a list
* eg : cdr of l where l is (hamburger) = ()

( cdr l) where l is ((x) t r) = (t r)

1. cons : adds an S - expression to the front of a list

* takes two arguments, second must be a list. output is a list
* eg : ( cons s l) where s is ((help) this) and l is (is very ((hard) to learn)) = (((help) this) is very ((hard) to learn))

1. null? : checks whether a list is empty list

* defined only for lists
* eg : ( null ? a) true or false where a is spaghetti = undefined, because you cannot ask for the null? of an atom

eg : (null? l) true or false where l is (a b c) = False

1. atom? : checks whether it is an atom

* takes one argument (any S - expression)
* eg : ( atom? s ) true or false where s is (Harry had a heap of apples) = True

eg : ( atom? (car ( cdr l))) true or false where l is (swing (low sweet) cherry oat) = False

1. eq? : checks if two non numeric atoms are equal

* eg : ( eq? ( car l) a ) true or false True, where l is (Mary had a little lamb chop) and a is Mary = True

eg : ( eq? n1 n2) true or false where n1 is 6 and n2 is 7 = undefined, cannot compare numbers

The Five Rules

1. The Law of Car : The primitive car is defined only for nonempty lists.
2. The Law of Cdr : The primitive cdr is defined only for nonempty lists. The cdr of any non-empty list

is always another list.

1. The Law of Cons : The primitive cons takes two arguments. The second argument to cons must be a list.

The result is a list.

1. The Law of Null? : The primitive null? is defined only for lists.
2. The Law of Eq? : The primitive eq? takes two arguments.Each must be a non-numeric atom.

Derived Functions

1. lat? : looks at each S - expression in a list and checks if it is an atom

(define lat?

(lambda (l)

(cond

((null ? l) #t )

(( atom? ( car l)) (lat ? ( cdr l)))

(else #f ))))

eg : (lat? l) where l is (bacon (and eggs)) = False

(lat ? l) where l is (bacon and eggs) = True

1. member? : checks if an atom belongs to a list of atoms

(define member?

(lambda (a lat)

(cond

((null ? lat) #f )

(else (or ( eq? ( car lat) a)

(member? a ( cdr lat)))))))

eg : ( member? a lat) where a is meat and lat is (potatoes and meat gravy) = True

Note : the second argument is a list of atoms and not any list!

Changing Primitive Names For Prompting And Constructing A Sample Prompt

car ---------> rac

cdr ---------> rdc

cons ---------> snoc

null? ---------> lun?

eq? ---------> qe?

atom? ---------> mota?

Given these primitives and the scope of the programming language Scheme, can an LLM reason to come up with functions like member? or lat? as needed by a given problem?

Let’s test this with some simple experiment!

Constructing The Prompt, Components :

1. Scope : Programming Language : Scheme

Manner : Recursive

Create functional & modular code

1. Primitives Definitions with Modified Names
2. A problem or task that would require the derived function
3. Output : Scheme code (parse output or instruct to output only code)

Simple Sample Prompt :

Prompt:

A new programming language follows the syntax of Scheme using keywords like lambda, define, cond, #t, #f and so on, and has the same basic data types : atom, list and S - expressions. Here are some basic functions in this programming language.

rac : gives first S-expression of a list

defined only for non - empty lists, returns a S - expression

rdc : gives list l without (by removing) rac l

defined only for non - empty lists, returns a list

snoc : adds an S - expression to the front of a list

takes two arguments, second must be a list. output is a list

lun? : checks whether a list is empty list

defined only for lists

mota? : checks whether it is an atom

takes one argument (any S - expression)

qe? : checks if two non numeric atoms are equal

I have two variables

a = meat

l = (mashed potatoes and meat gravy)

I want to check if a is in the list of atoms l. Can you help me write the code for this? Output only the code directly. Let's make it a function so that I just have to pass a and l to it.

Output:

Here is the code:

```

(define (member? a l)

(cond

((lun? l) #f)

((qe? a (rac l)) #t)

(else (member? a (rdc l)))))

```

Equivalently, fixing syntax,

(define member?

(lambda (a l)

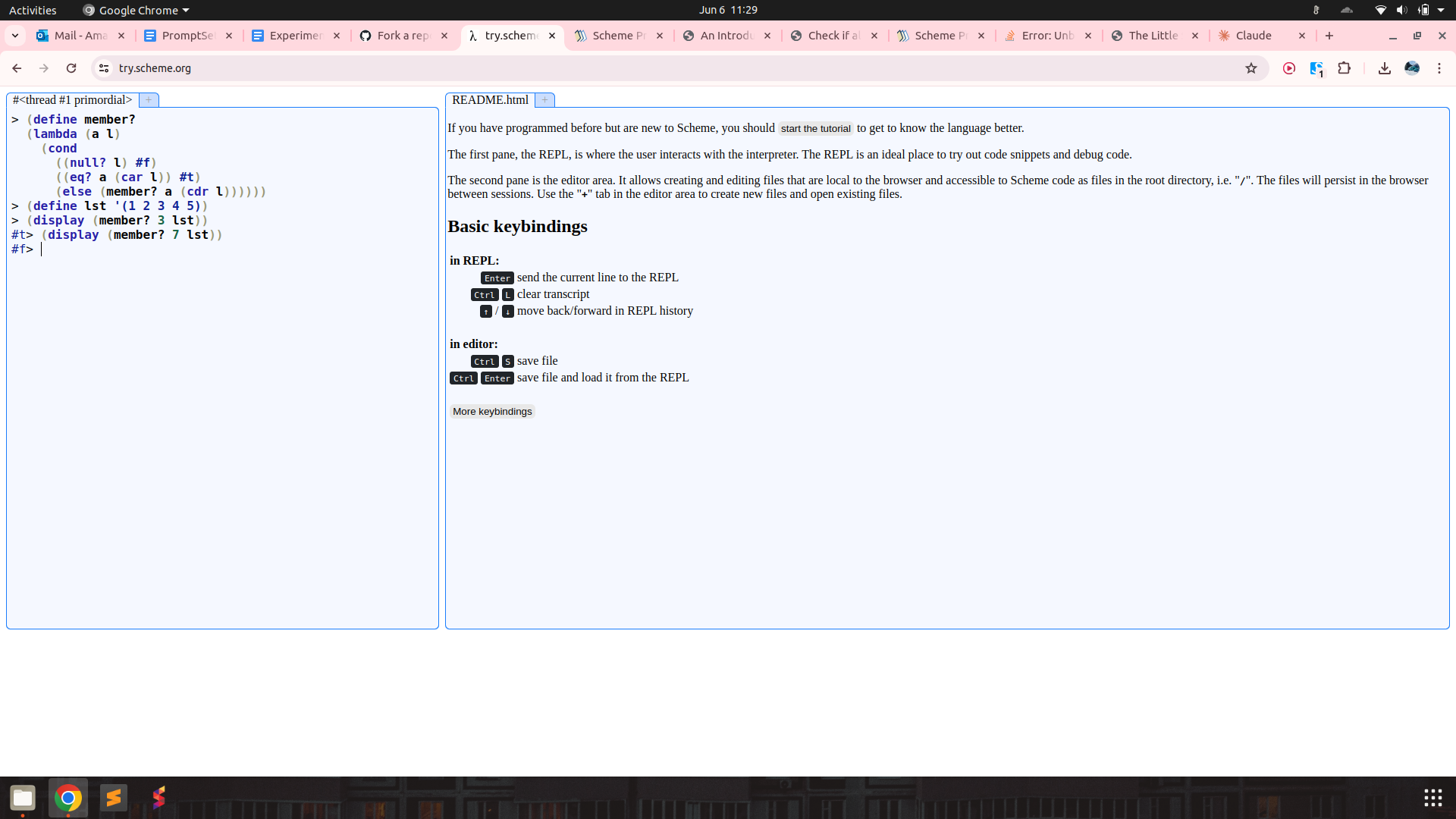
(cond

((null? l) #f)

((eq? a (car l)) #t)

(else (member? a (cdr l))))))

Testing Using Scheme : function works!



Let’s try and remove any mention of the language Scheme and instead describe its syntax.

Prompt :

The programming language we're discussing has a unique syntax that is centred around functional programming and recursion. Here's a breakdown of some key elements:

atoms: An atom is a single, indivisible value such as a number, boolean, or symbol.

lists: A list is a compound data structure that contains an ordered sequence of elements, which can be atoms or other lists.

S-expressions (Symbolic Expressions): S-expressions are the primary syntactic constructs in the language, consisting of atoms or lists enclosed in parentheses.

define: This keyword is used to define variables, functions, or procedures. For example:

lambda: This keyword is used to create anonymous functions or lambda expressions. It takes a list of arguments and a body of expressions to be evaluated.

cond: This keyword is used for conditional branching. It evaluates a series of test expressions and evaluates the corresponding expression for the first test that evaluates to true.

else: This keyword is used in the cond expression to specify the expression to be evaluated if none of the previous test expressions are true.

'(): This represents the empty list, often used as a terminating condition in recursive list operations.

#t and #f: These represent the boolean values true and false, respectively.

The language uses prefix notation, where the operator or function comes before its operands or arguments, enclosed in parentheses. It also supports recursion, where the function calls itself with a smaller portion of the list until the base case is reached.

Here are some basic functions in this programming language.

rac : gives first S-expression of a list

defined only for non - empty lists, returns a S - expression

rdc : gives list l without (by removing) rac l

defined only for non - empty lists, returns a list

snoc : adds an S - expression to the front of a list

takes two arguments, second must be a list. output is a list

lun? : checks whether a list is empty list

defined only for lists

mota? : checks whether it is an atom

takes one argument (any S - expression)

qe? : checks if two non numeric atoms are equal

I have two variables

a = meat

l = (mashed potatoes and meat gravy)

I want to check if a is in the list of atoms l. Can you help me write the code for this? Output only the code directly.

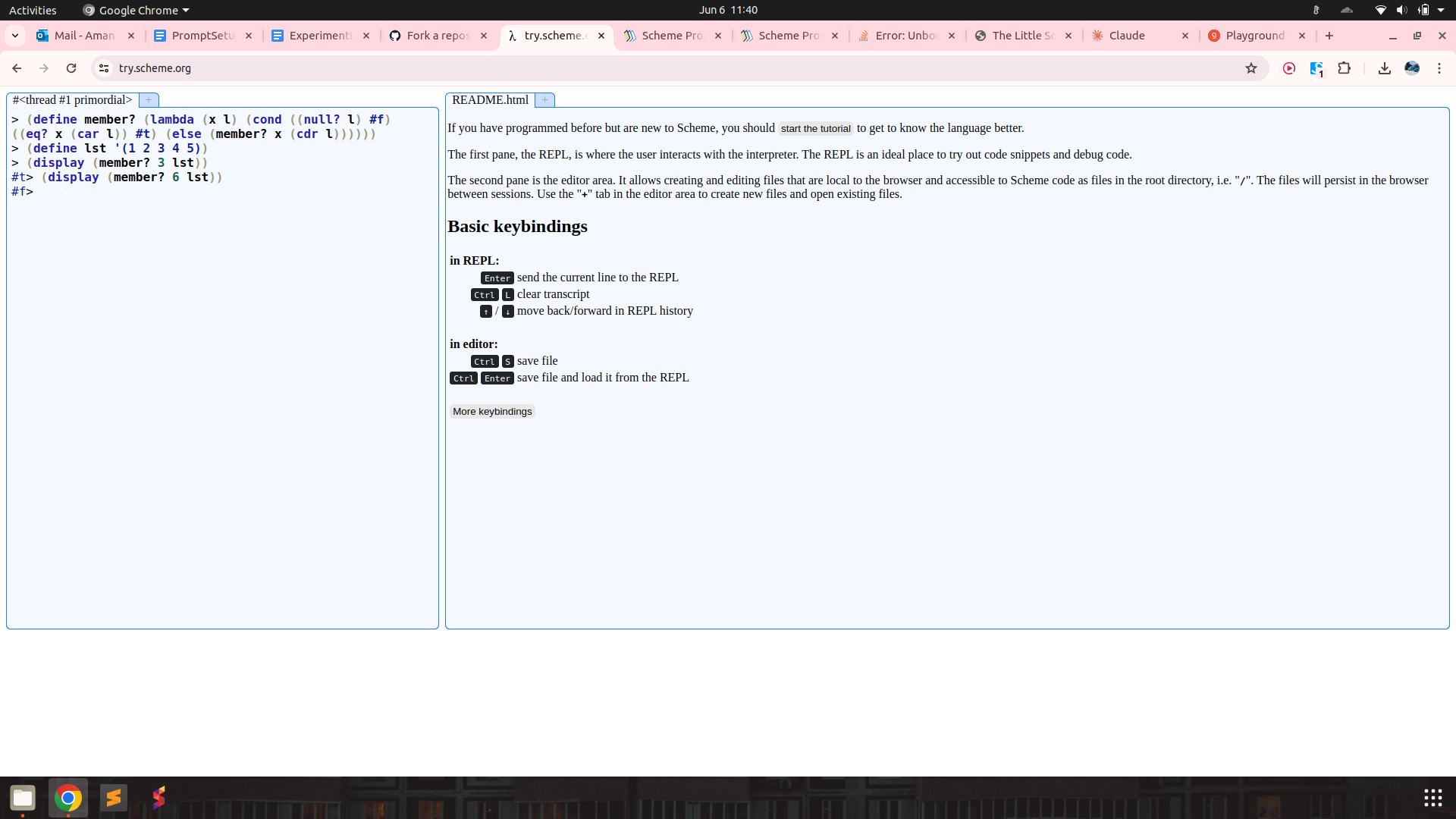
Output :

(define member? (lambda (x l) (cond ((lun? l) #f) ((qe? x (rac l)) #t) (else (member? x (rdc l))))))

(member? a l)

Result :

Decoding primitive names back to original and testing, function works as needed.



Checking if a list l is a list of atoms :

The programming language we're discussing has a unique syntax that is centred around functional programming and recursion. Here's a breakdown of some key elements:

atoms: An atom is a single, indivisible value such as a number, boolean, or symbol.

lists: A list is a compound data structure that contains an ordered sequence of elements, which can be atoms or other lists.

S-expressions (Symbolic Expressions): S-expressions are the primary syntactic constructs in the language, consisting of atoms or lists enclosed in parentheses.

define: This keyword is used to define variables, functions, or procedures. For example:

lambda: This keyword is used to create anonymous functions or lambda expressions. It takes a list of arguments and a body of expressions to be evaluated.

cond: This keyword is used for conditional branching. It evaluates a series of test expressions and evaluates the corresponding expression for the first test that evaluates to true.

else: This keyword is used in the cond expression to specify the expression to be evaluated if none of the previous test expressions are true.

'(): This represents the empty list, often used as a terminating condition in recursive list operations.

#t and #f: These represent the boolean values true and false, respectively.

The language uses prefix notation, where the operator or function comes before its operands or arguments, enclosed in parentheses. It also supports recursion, where the function calls itself with a smaller portion of the list until the base case is reached.

Here are some basic functions in this programming language.

rac : gives first S-expression of a list

defined only for non - empty lists, returns a S - expression

rdc : gives list l without (by removing) rac l

defined only for non - empty lists, returns a list

snoc : adds an S - expression to the front of a list

takes two arguments, second must be a list. output is a list

lun? : checks whether a list is empty list

defined only for lists

mota? : checks whether it is an atom

takes one argument (any S - expression)

qe? : checks if two non numeric atoms are equal

I have a list

l = (mashed potatoes and meat gravy)

I want to check if l is a list of atoms. Can you help me write the code for this? Output only the code directly.

Output :

(define atoms? (lambda (l) (cond ((lun? l) #t) ((mota? (rac l)) (atoms? (rdc l))) (else #f))))

Result :

(define atoms? (lambda (l) (cond ((null? l) #t) ((atom? (car l)) (atoms? (cdr l))) (else #f))))

function is correct.

Finding index of element in a list

Prompt :

The programming language we're discussing has a unique syntax that is centred around functional programming and recursion. Here's a breakdown of some key elements:

atoms: An atom is a single, indivisible value such as a number, boolean, or symbol.

lists: A list is a compound data structure that contains an ordered sequence of elements, which can be atoms or other lists.

S-expressions (Symbolic Expressions): S-expressions are the primary syntactic constructs in the language, consisting of atoms or lists enclosed in parentheses.

define: This keyword is used to define variables, functions, or procedures. For example:

lambda: This keyword is used to create anonymous functions or lambda expressions. It takes a list of arguments and a body of expressions to be evaluated.

cond: This keyword is used for conditional branching. It evaluates a series of test expressions and evaluates the corresponding expression for the first test that evaluates to true.

else: This keyword is used in the cond expression to specify the expression to be evaluated if none of the previous test expressions are true.

'(): This represents the empty list, often used as a terminating condition in recursive list operations.

#t and #f: These represent the boolean values true and false, respectively.

The language uses prefix notation, where the operator or function comes before its operands or arguments, enclosed in parentheses. It also supports recursion, where the function calls itself with a smaller portion of the list until the base case is reached.

Here are some basic functions in this programming language.

rac : gives first S-expression of a list

defined only for non - empty lists, returns a S - expression

rdc : gives list l without (by removing) rac l

defined only for non - empty lists, returns a list

snoc : adds an S - expression to the front of a list

takes two arguments, second must be a list. output is a list

lun? : checks whether a list is empty list

defined only for lists

mota? : checks whether it is an atom

takes one argument (any S - expression)

qe? : checks if two non numeric atoms are equal

I have two variables

a = meat

l = (mashed potatoes and meat gravy)

I want to check if a is in the list of atoms l. If yes, return its index, if not return -1. Can you help me write the code for this? Output only the code directly.

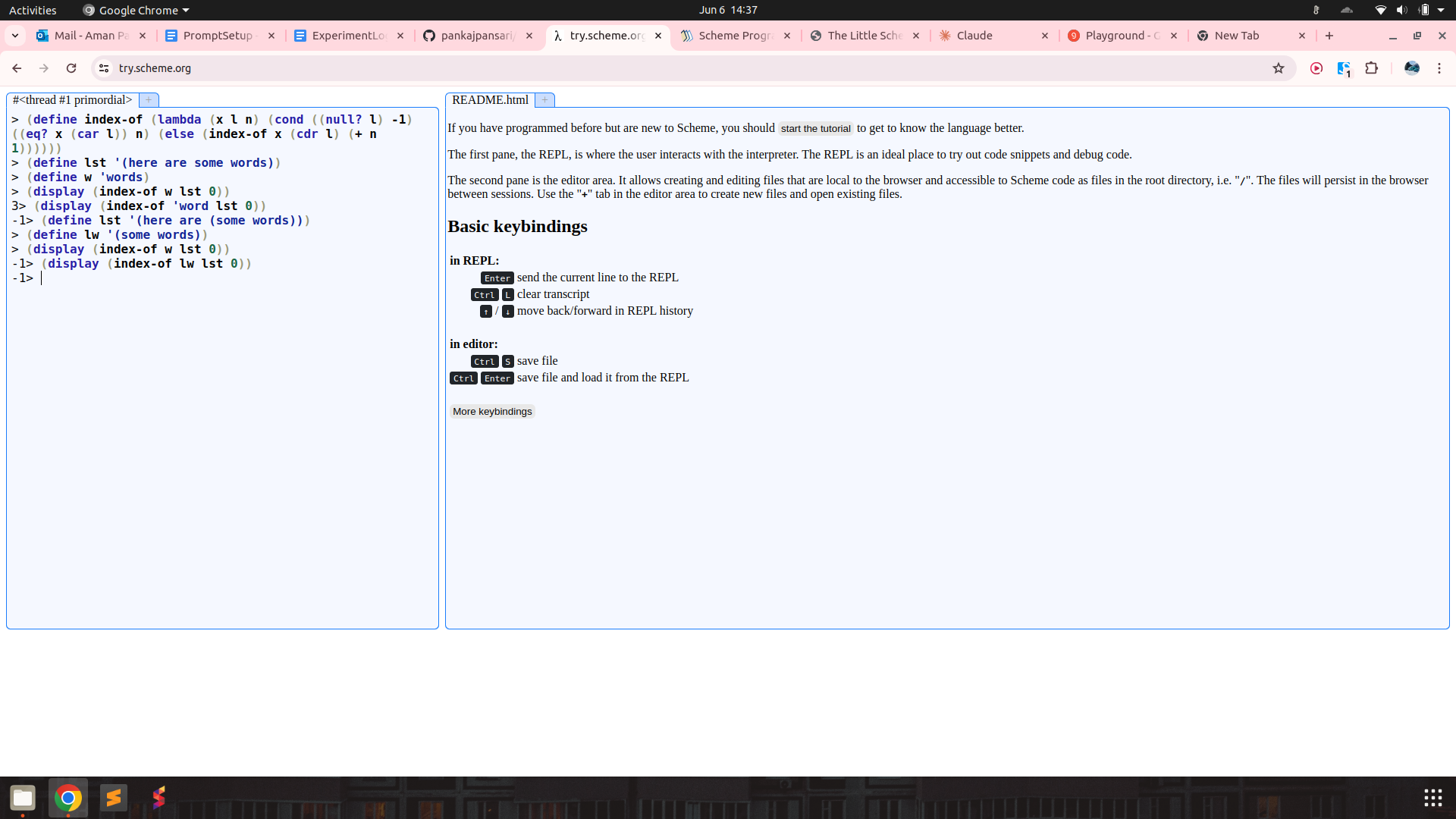
Output :

(define index-of (lambda (x l n) (cond ((lun? l) -1) ((qe? x (rac l)) n) (else (index-of x (rdc l) (+ n 1))))))

Result :

Works as needed for list of atoms. Also takes a third argument, n of starting index. (for eg n = 0 by default in python).

However, does not work for lists containing other lists!



Counting Number of Occurrences in a List

Prompt :

The programming language we're discussing has a unique syntax that is centred around functional programming and recursion. Here's a breakdown of some key elements:

atoms: An atom is a single, indivisible value such as a number, boolean, or symbol.

lists: A list is a compound data structure that contains an ordered sequence of elements, which can be atoms or other lists.

S-expressions (Symbolic Expressions): S-expressions are the primary syntactic constructs in the language, consisting of atoms or lists enclosed in parentheses.

define: This keyword is used to define variables, functions, or procedures. For example:

lambda: This keyword is used to create anonymous functions or lambda expressions. It takes a list of arguments and a body of expressions to be evaluated.

cond: This keyword is used for conditional branching. It evaluates a series of test expressions and evaluates the corresponding expression for the first test that evaluates to true.

else: This keyword is used in the cond expression to specify the expression to be evaluated if none of the previous test expressions are true.

'(): This represents the empty list, often used as a terminating condition in recursive list operations.

#t and #f: These represent the boolean values true and false, respectively.

The language uses prefix notation, where the operator or function comes before its operands or arguments, enclosed in parentheses. It also supports recursion, where the function calls itself with a smaller portion of the list until the base case is reached.

Here are some basic functions in this programming language.

rac : gives first S-expression of a list

defined only for non - empty lists, returns a S - expression

rdc : gives list l without (by removing) rac l

defined only for non - empty lists, returns a list

snoc : adds an S - expression to the front of a list

takes two arguments, second must be a list. output is a list

lun? : checks whether a list is empty list

defined only for lists

mota? : checks whether it is an atom

takes one argument (any S - expression)

qe? : checks if two non numeric atoms are equal

I have two variables

a = meat

l = (mashed potatoes and meat gravy)

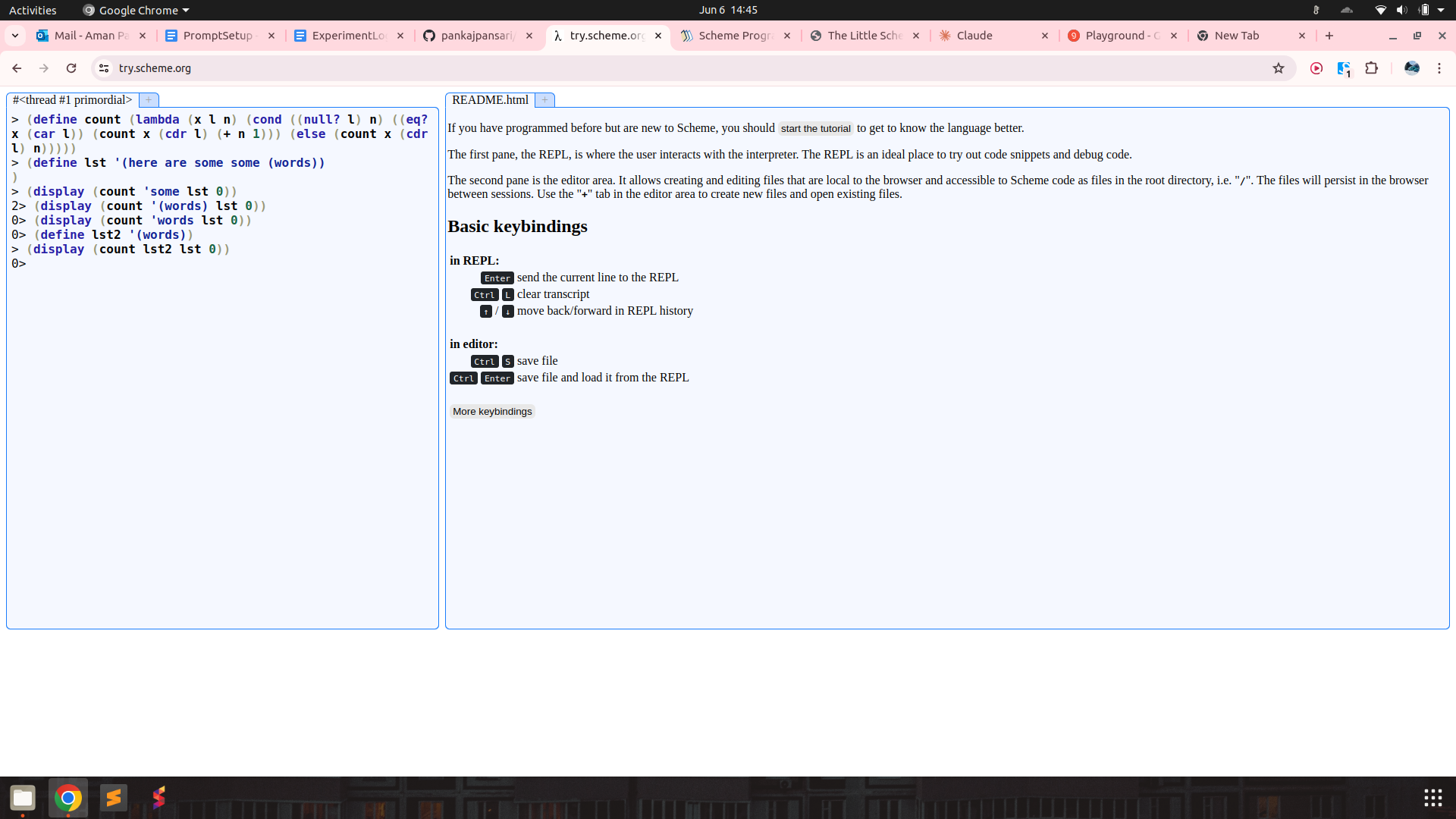
I want to count how many times a occurs in the list l. a could even be a list and the list l can have other lists as its elements. Can you help me write the code for this? Output only the code directly.

Output :

(define count-occurrences (lambda (x l n) (cond ((lun? l) n) ((qe? x (rac l)) (count-occurrences x (rdc l) (+ n 1))) (else (count-occurrences x (rdc l) n)))))

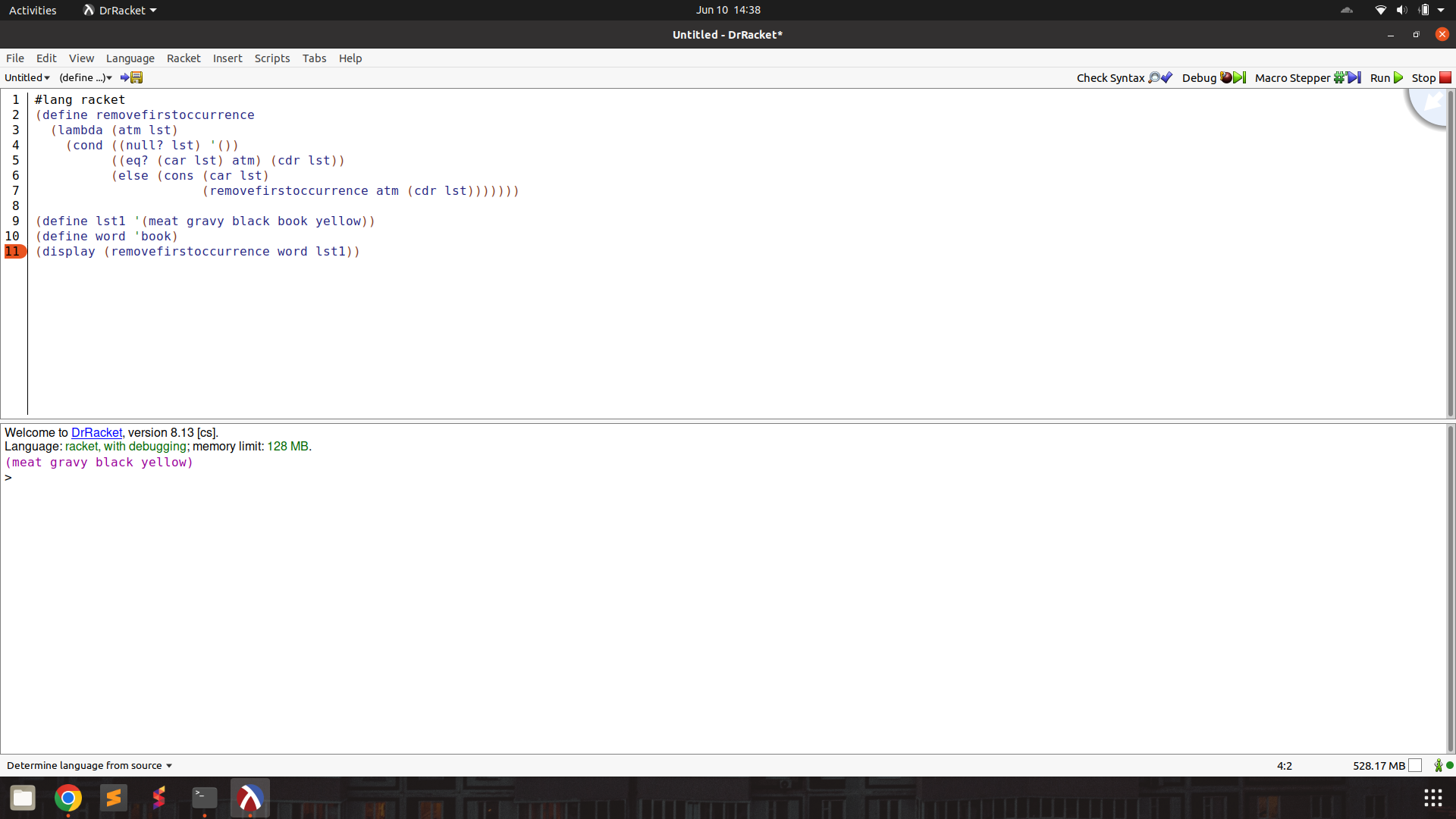
Result :

Works for list of atoms again, but fails for nested lists. Prompt also specifies that lists may be nested.



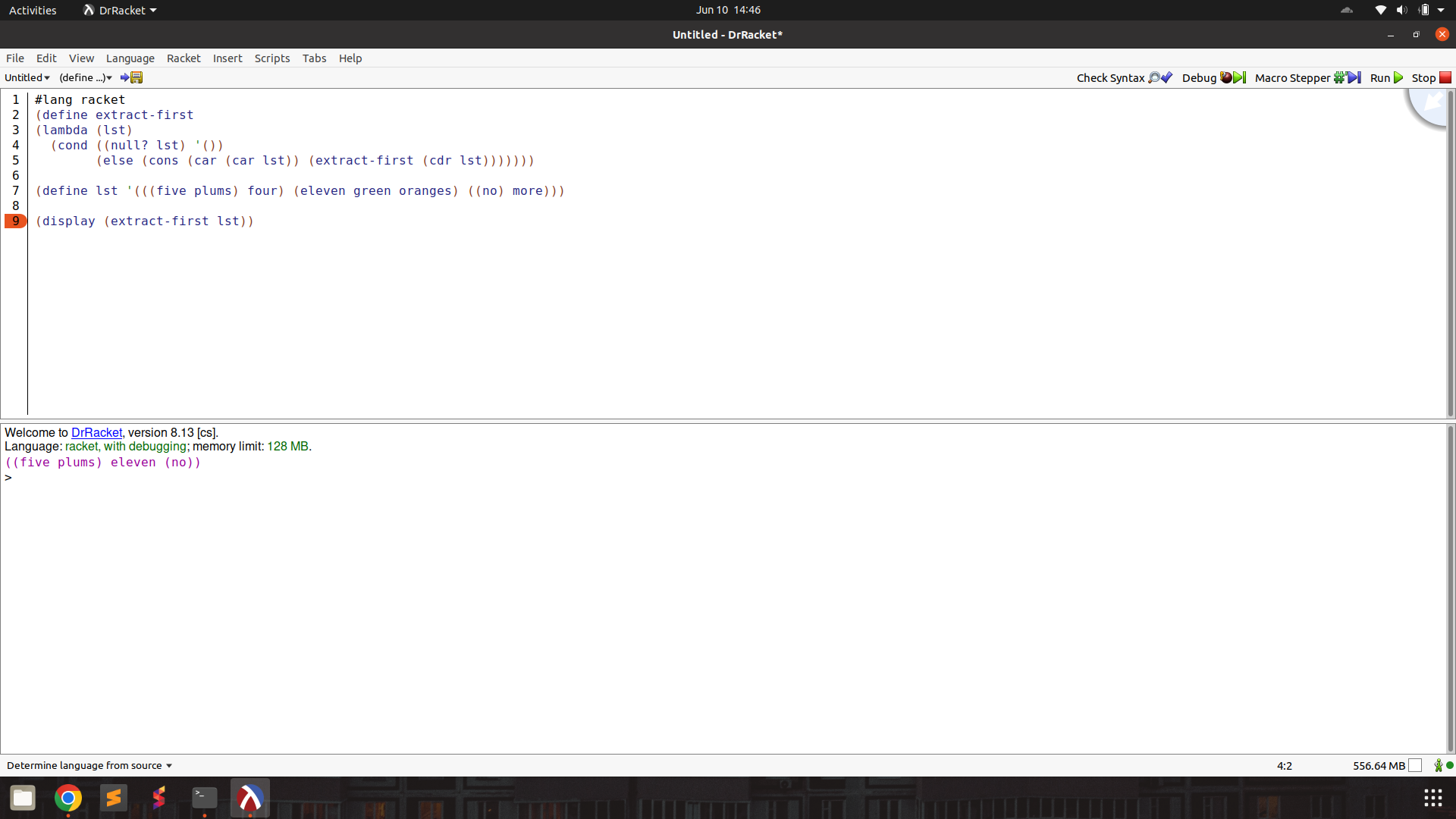
rember

result :



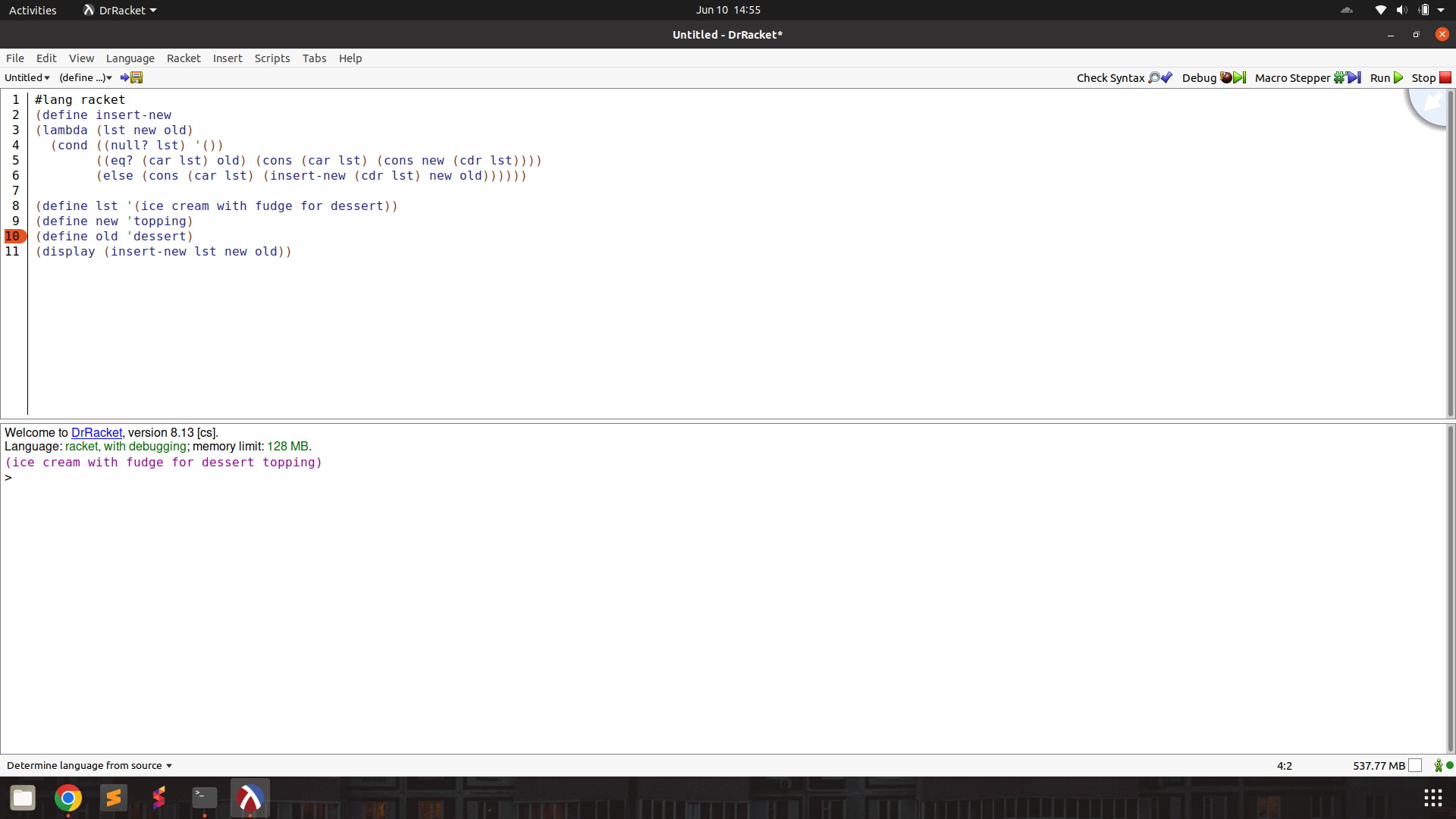
firsts

result : function works!



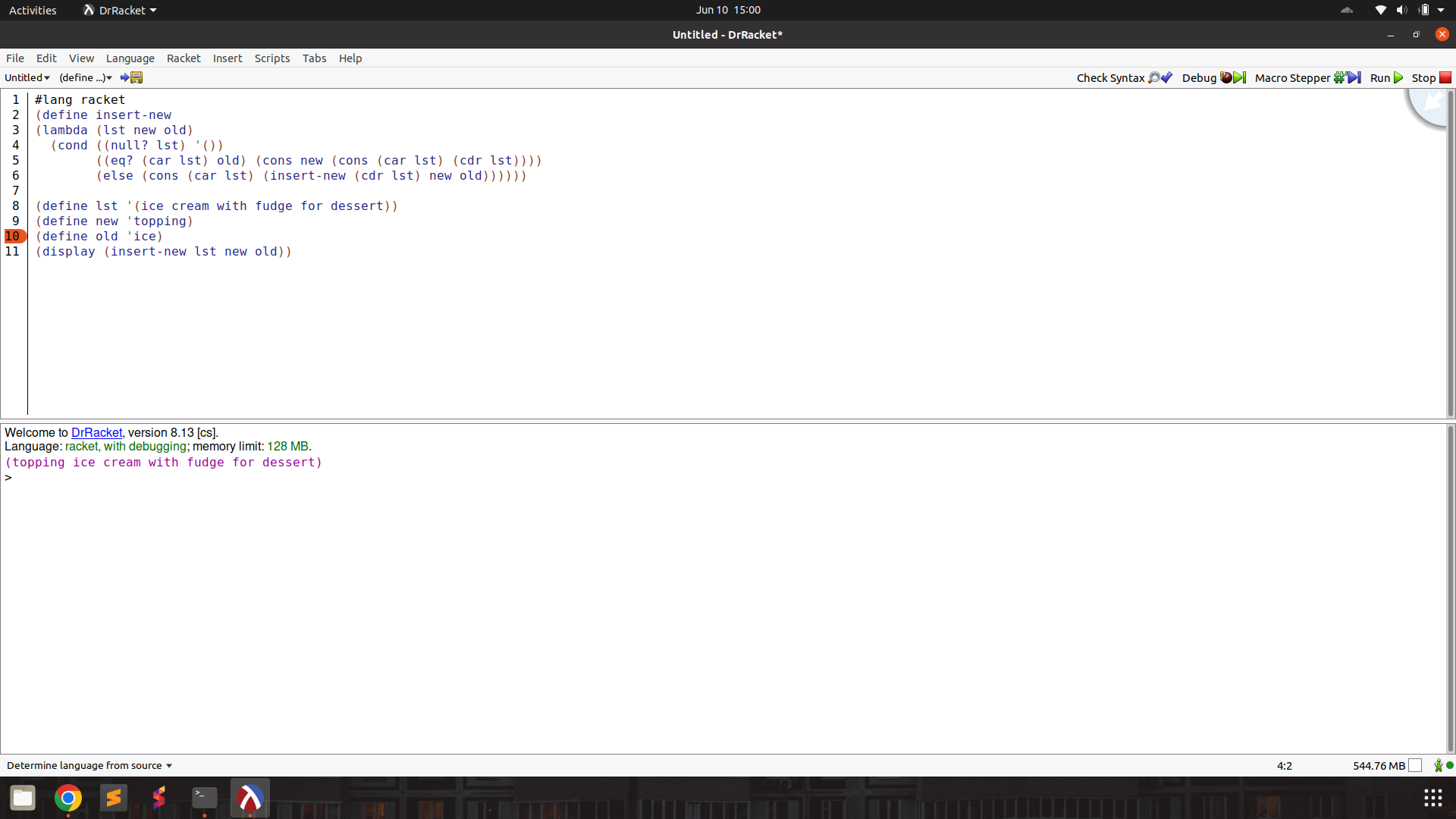
insertr

result : function works! (only syntax changes, lambda keyword not used)



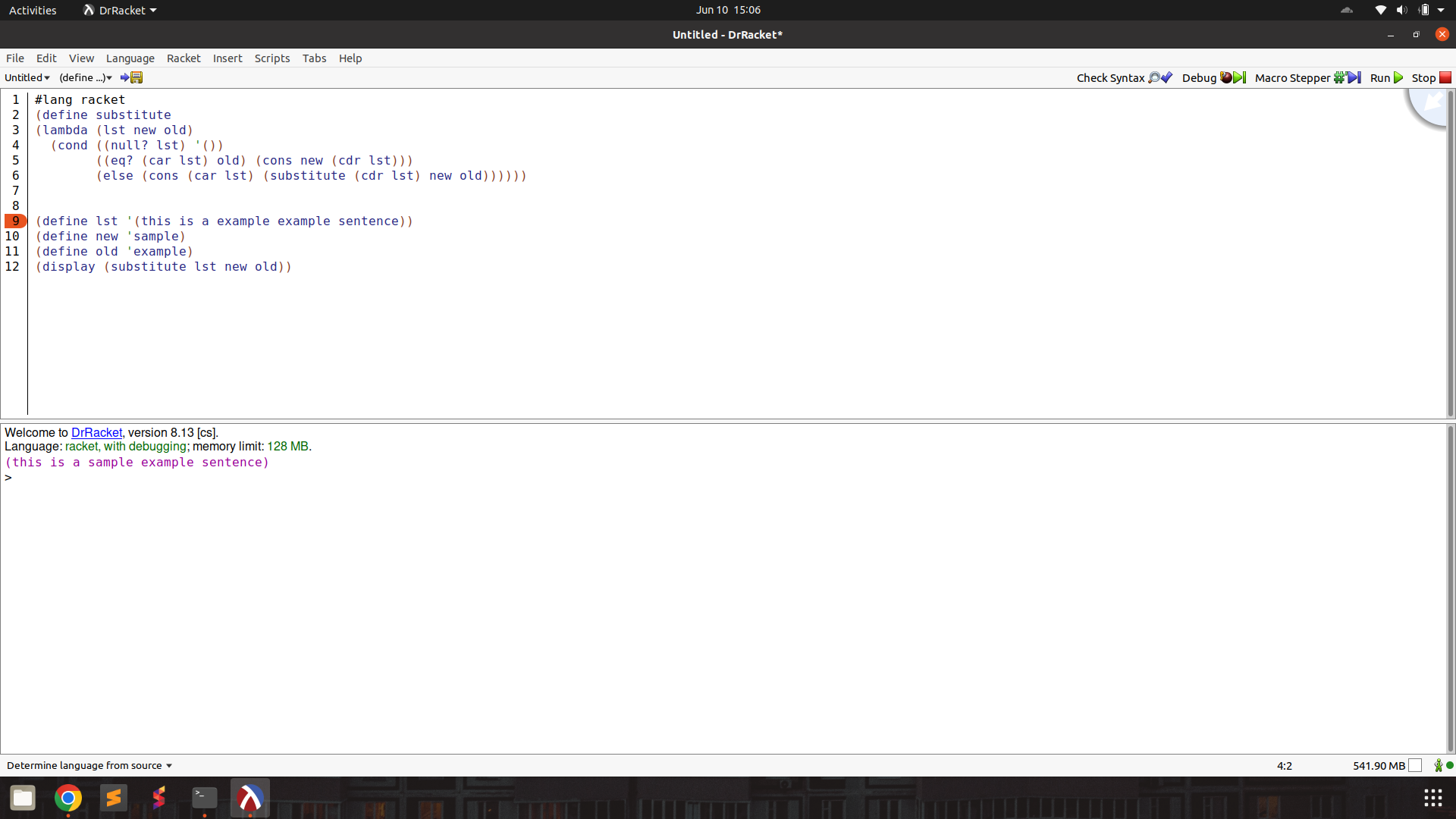
insertl

result : function works! (only syntax changes, lambda keyword not used)



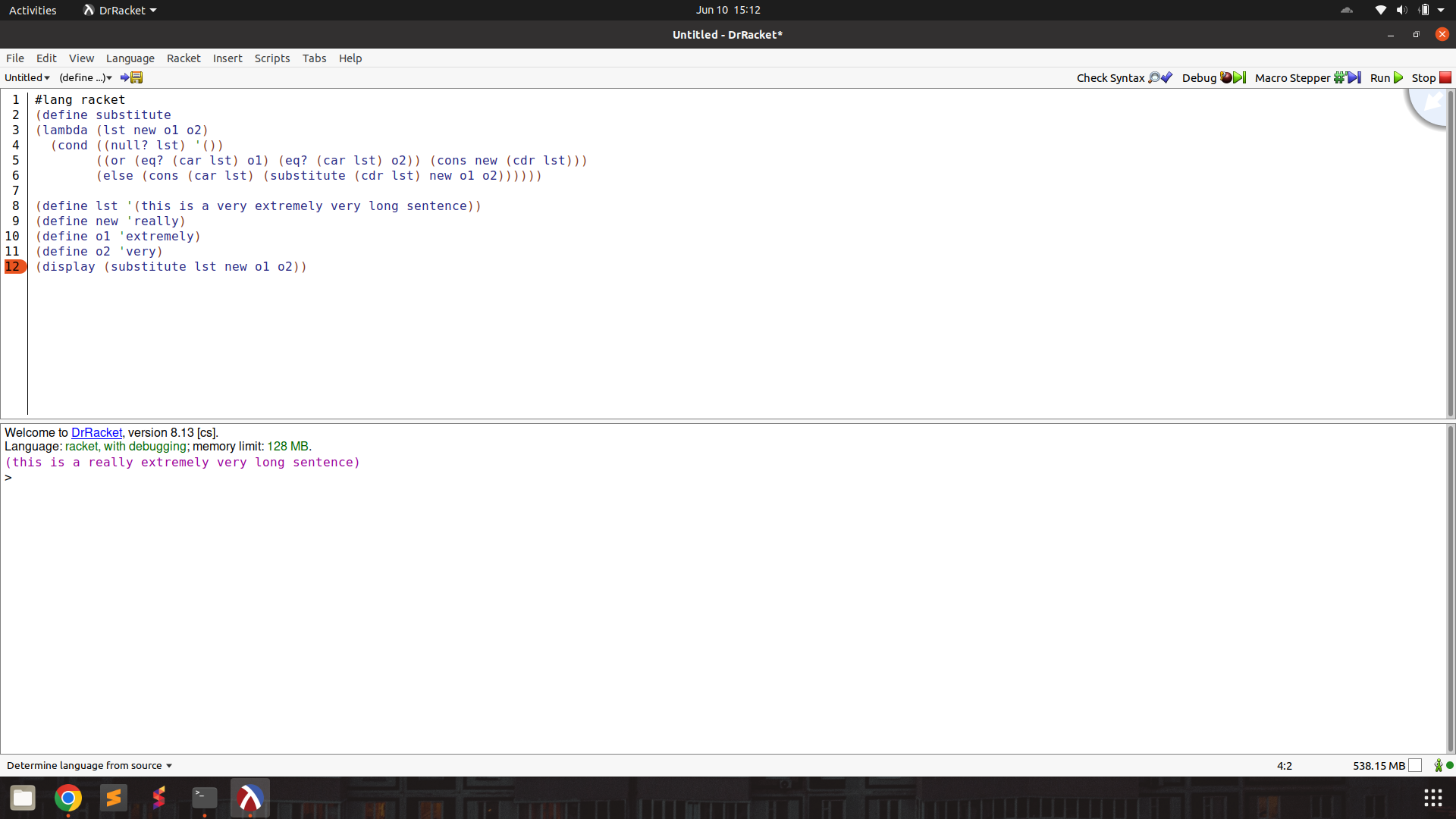
subst

result : function works when syntax is fixed



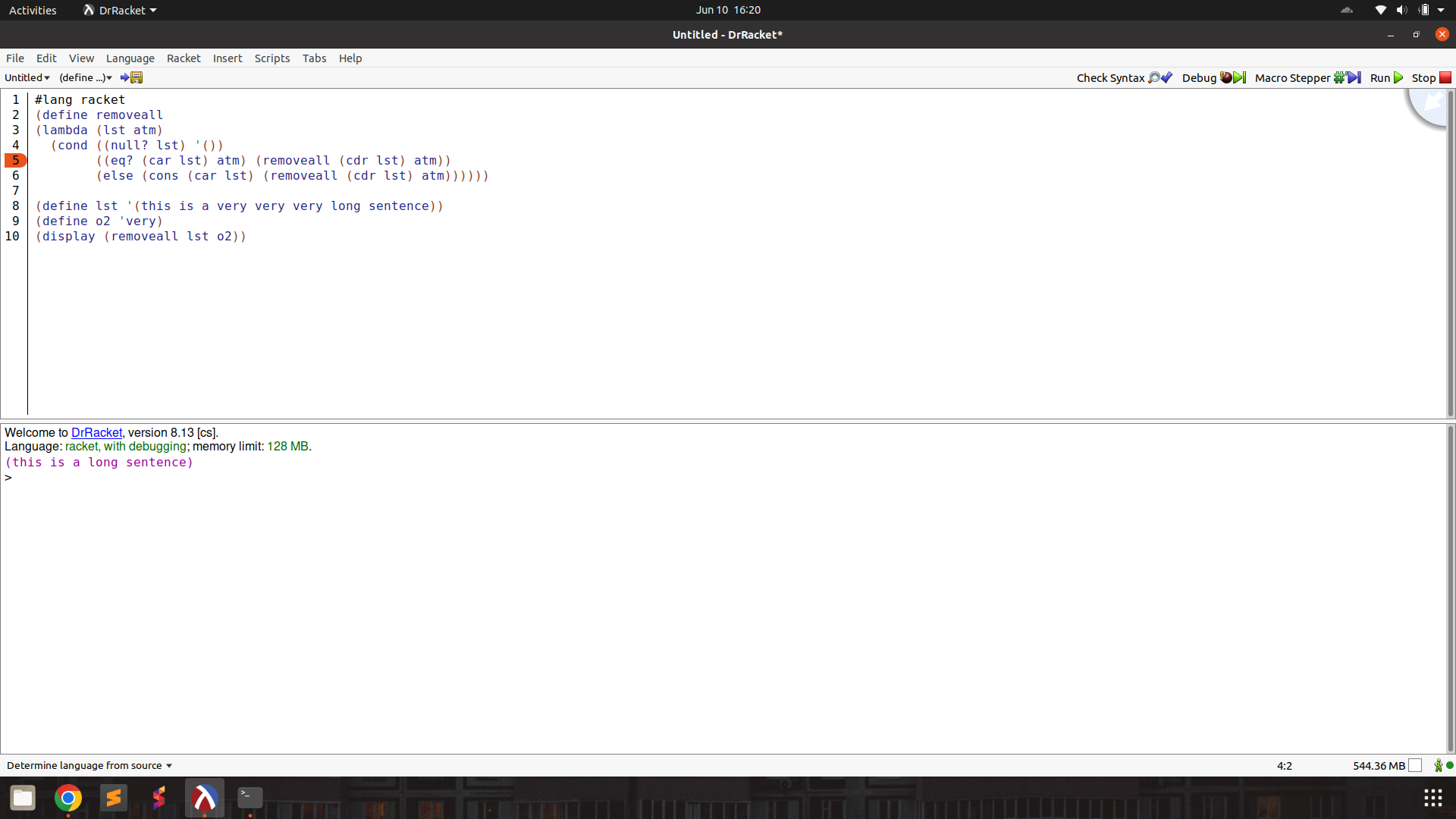
subst2

result : function works when syntax is fixed!



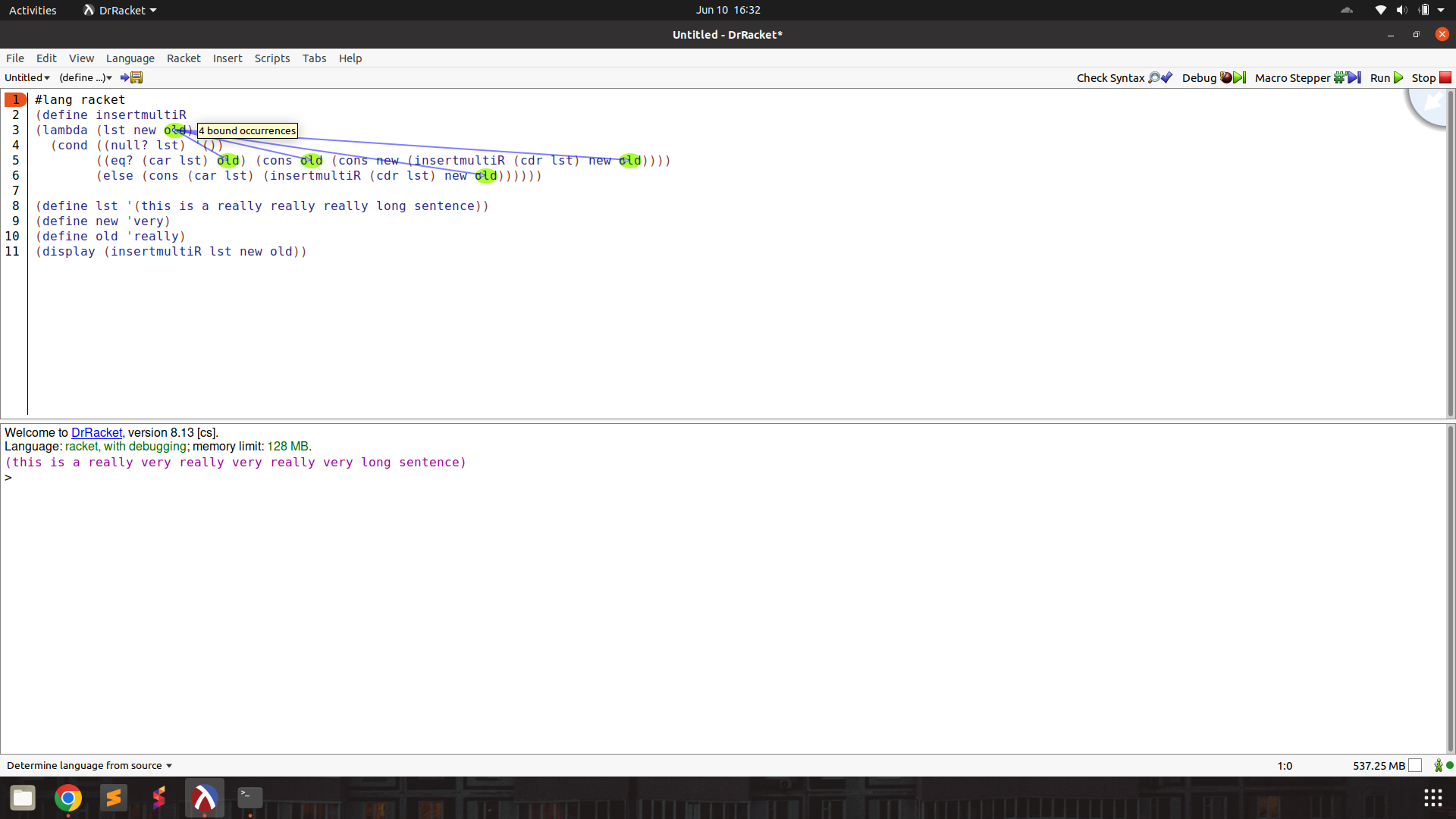
multirember

result : function works when syntax and identifier name is fixed



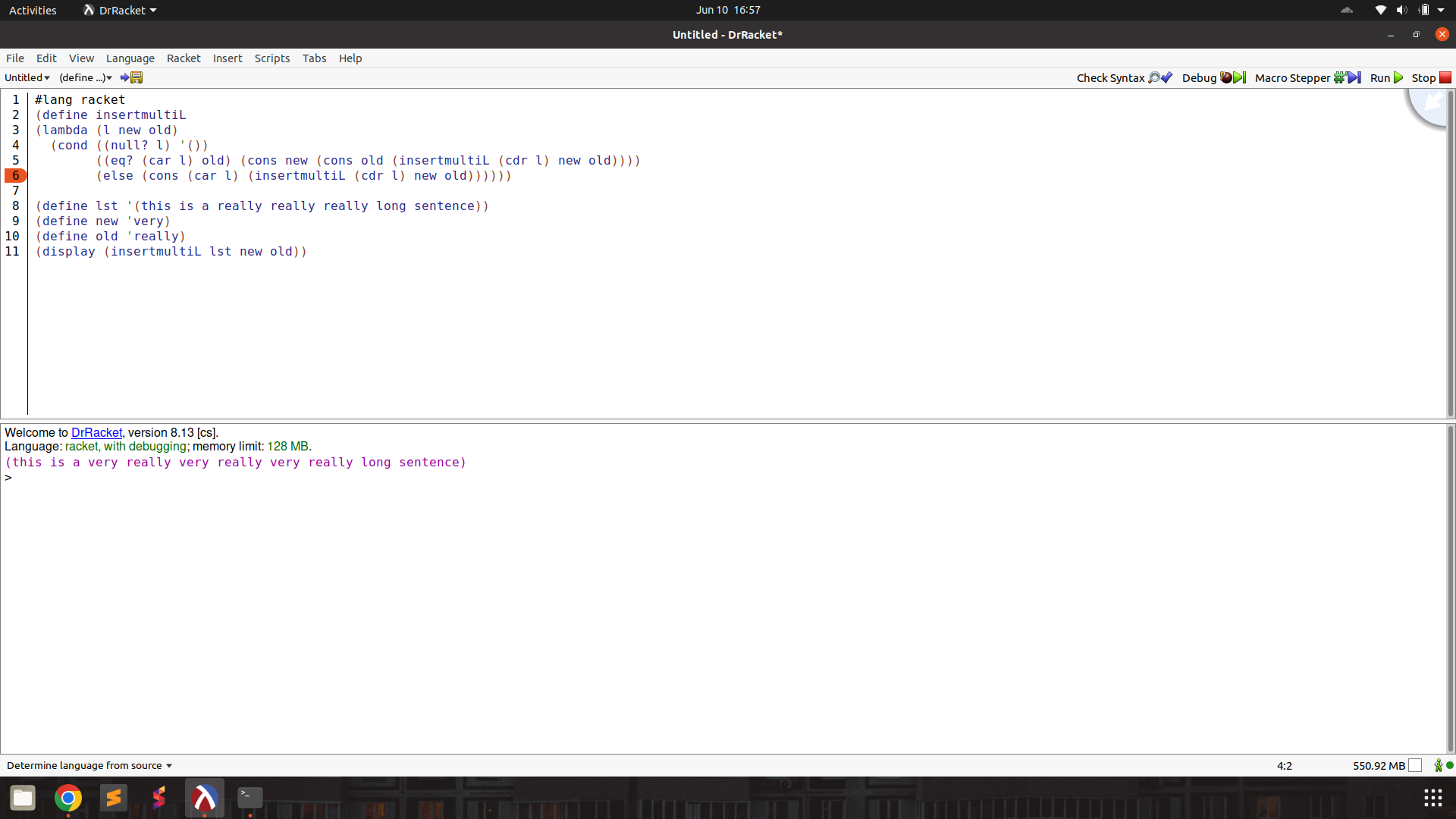
multiinsertR

result : function works as expected after syntax changes



multiinsertL

result : works as expected, usual syntax changes



multisubst

result : fails as atom? primitive seems to be missing in most implementations of Scheme

