**Numbers**

add1 & sub1 :

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 variables

n = 6

m = 5

I want to write functions for numbers, let's start with functions add1 and subtract1. They do what it sounds like, add or subtract one from a number! For numbers, the equivalent of lun? is orez?

which checks if a number is zero.

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

output :

Here are the functions `add1` and `subtract1`:

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

test : decoding primitive name and testing, functions work.



add & subtract :

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 want to write functions for numbers, to add and subtract them. For numbers, the equivalent of lun? is orez?

which checks if a number is zero. The functions add1 and subtract1 can be used, they are defined as below  
  
(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

output :

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

(define subtract (lambda (x y)

(cond ((orez? x) (cond ((orez? y) 0)

(else (subtract (subtract1 x) (subtract1 y)))))

(else (subtract (subtract1 x) y)))))

test : decoding primitive names and testing,

(define add (lambda (x y)

(cond ((zero? x) y)

(else (add (subtract1 x) (add1 y))))))

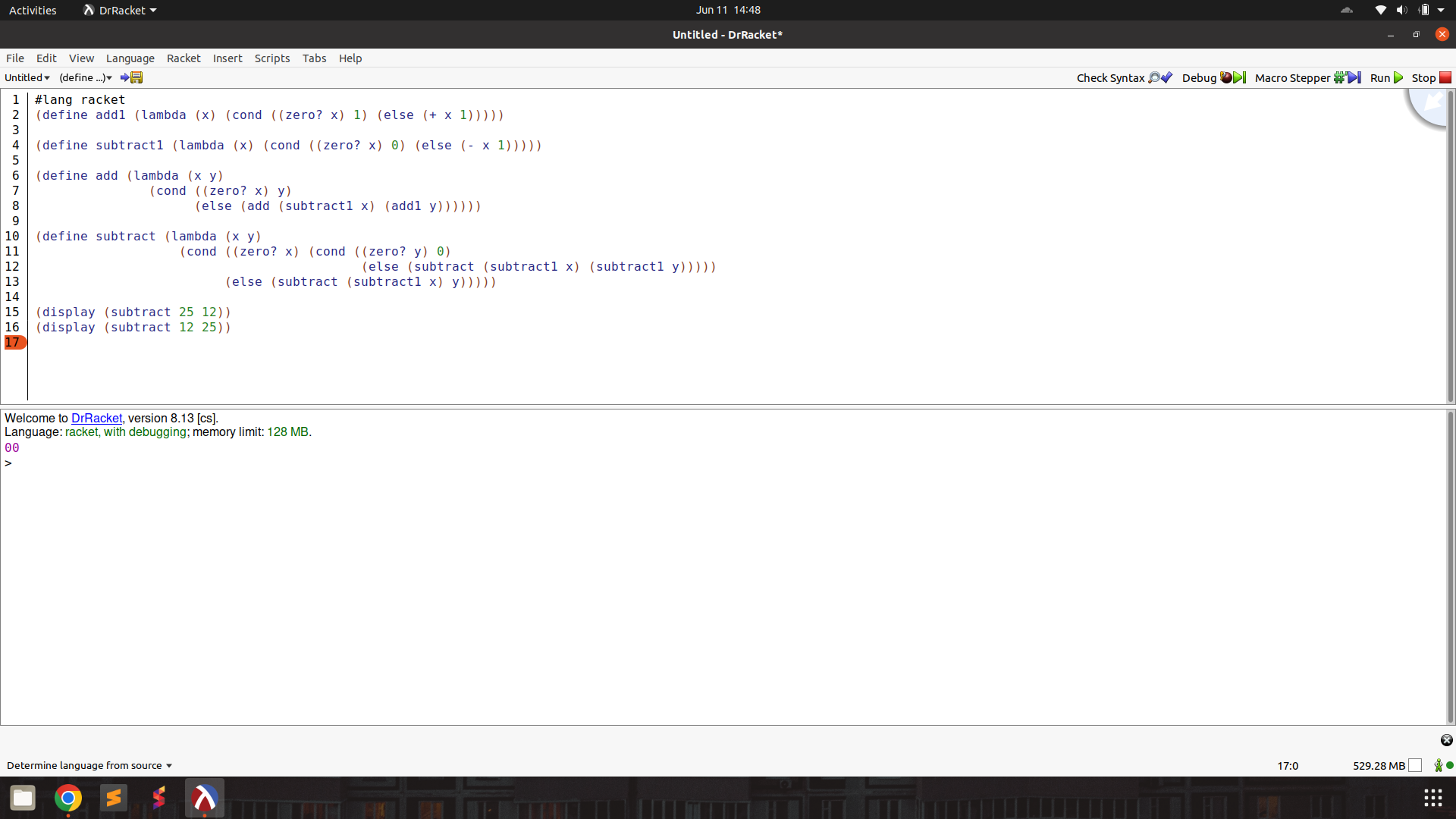
(define subtract (lambda (x y)

(cond ((zero? x) (cond ((zero? y) 0)

(else (subtract (subtract1 x) (subtract1 y)))))

(else (subtract (subtract1 x) y)))))

Add function is correct, however subtract is logically wrong and always returns 0. Multiple inferences sampled, with same result.



addtup :

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

function add is defined as

(define add (lambda (x y)

(cond ((zero? x) y)

(else (add (subtract1 x) (add1 y))))))

I have a tuple

tup = (25 89 34 19)

I want to write a function to add all the numbers in my tuple. Use the existing functions and primitives

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

output :

(define sum-list (lambda (l)

(cond ((lun? l) 0)

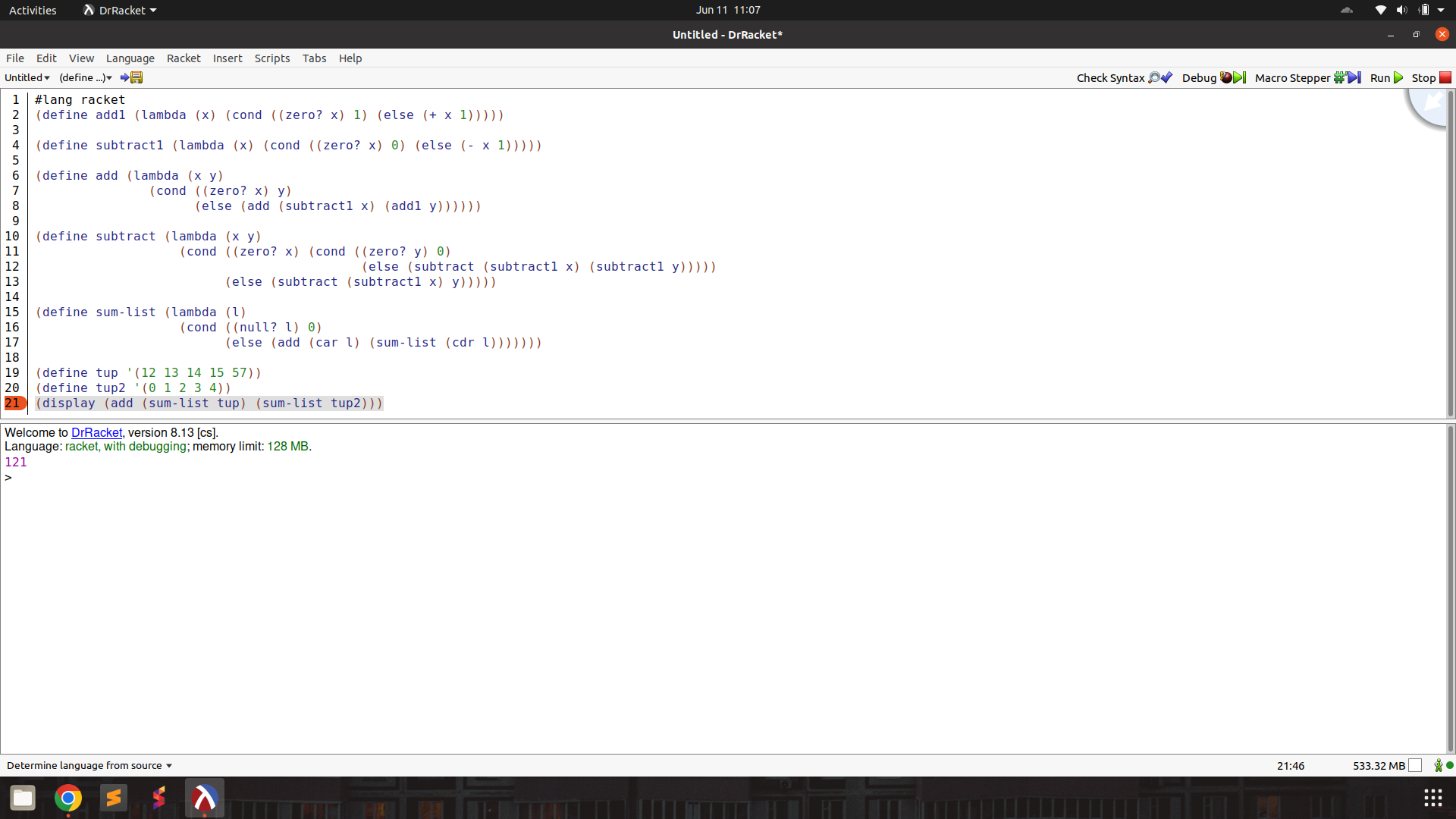
(else (add (rac l) (sum-list (rdc l)))))))

test : works correctly!

(define sum-list (lambda (l)

(cond ((null? l) 0)

(else (add (car l) (sum-list (cdr l)))))))



Note :

Model : LLaMA3-70b (temp. = 0.7), code on Colab

subtract

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 want to write functions for numbers to subtract them. For numbers, the equivalent of lun? is orez?

which checks if a number is zero. The functions add1, add and subtract1 have already been defined and can be used.

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

Output :

(define subtract (lambda (x y)

(cond ((orez? y) x)

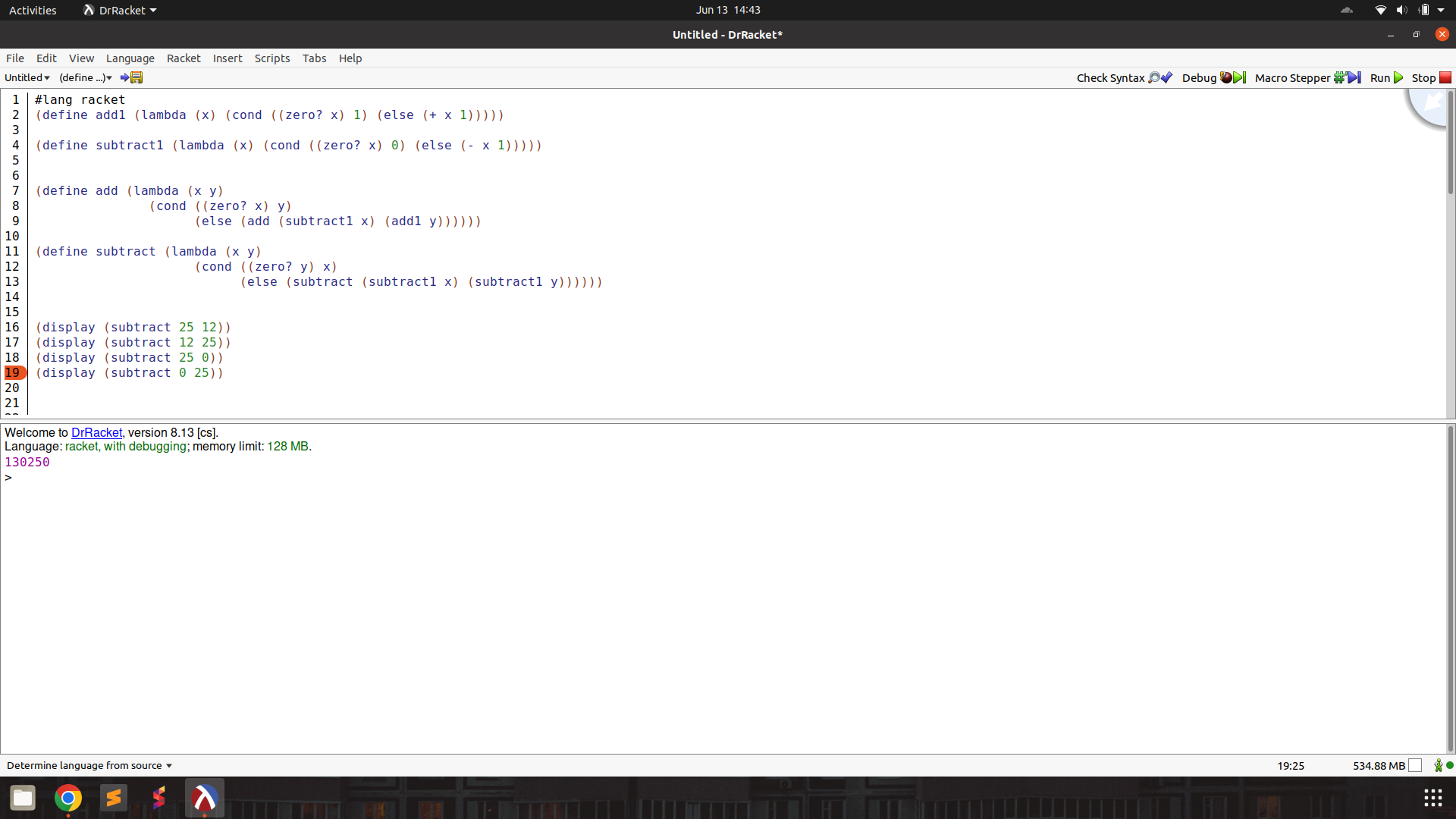
(else (subtract (subtract1 x) (subtract1 y))))))

Test : Works!.

(define subtract (lambda (x y)

(cond ((zero? y) x)

(else (subtract (subtract1 x) (subtract1 y))))))



mult

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 want to write functions for numbers to multiply two numbers. For numbers, the equivalent of lun? is orez?

which checks if a number is zero. The functions add1, add, subtract1 and subtract have already been defined and can be used.

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

(define subtract (lambda (x y)

(cond ((orez? y) x)

(else (subtract (subtract1 x) (subtract1 y))))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

Output :

(define multiply (lambda (x y)

(cond ((orez? x) 0)

((orez? y) 0)

(else (add (multiply (subtract1 x) y) y)))))

Test : function works!

(define multiply (lambda (x y)

(cond ((zero? x) 0)

((zero? y) 0)

(else (add (multiply (subtract1 x) y) y)))))



tup+ : add two tuples elementwise

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 want to write functions for numbers and tuple. Given two tuples of numbers, I want to add the numbers position wise. The tuples may be of unequal lengths. Here is an example,

tup1 is (3 7 8 1) and tup2 is (4 6), then my function should return (7 13 8 1).

For numbers, the equivalent of lun? is orez?

which checks if a number is zero. The functions add1, add, subtract1 and subtract have already been defined and can be used.

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

(define subtract (lambda (x y)

(cond ((orez? y) x)

(else (subtract (subtract1 x) (subtract1 y))))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

Output :

Here is the code:

```

(define add-tuples (lambda (t1 t2)

(cond ((lun? t1) t2)

((lun? t2) t1)

(else (snoc (add (rac t1) (rac t2))

(add-tuples (rdc t1) (rdc t2)))))))

```

Test : function works! output needs to be cleaned and primitive names decoded

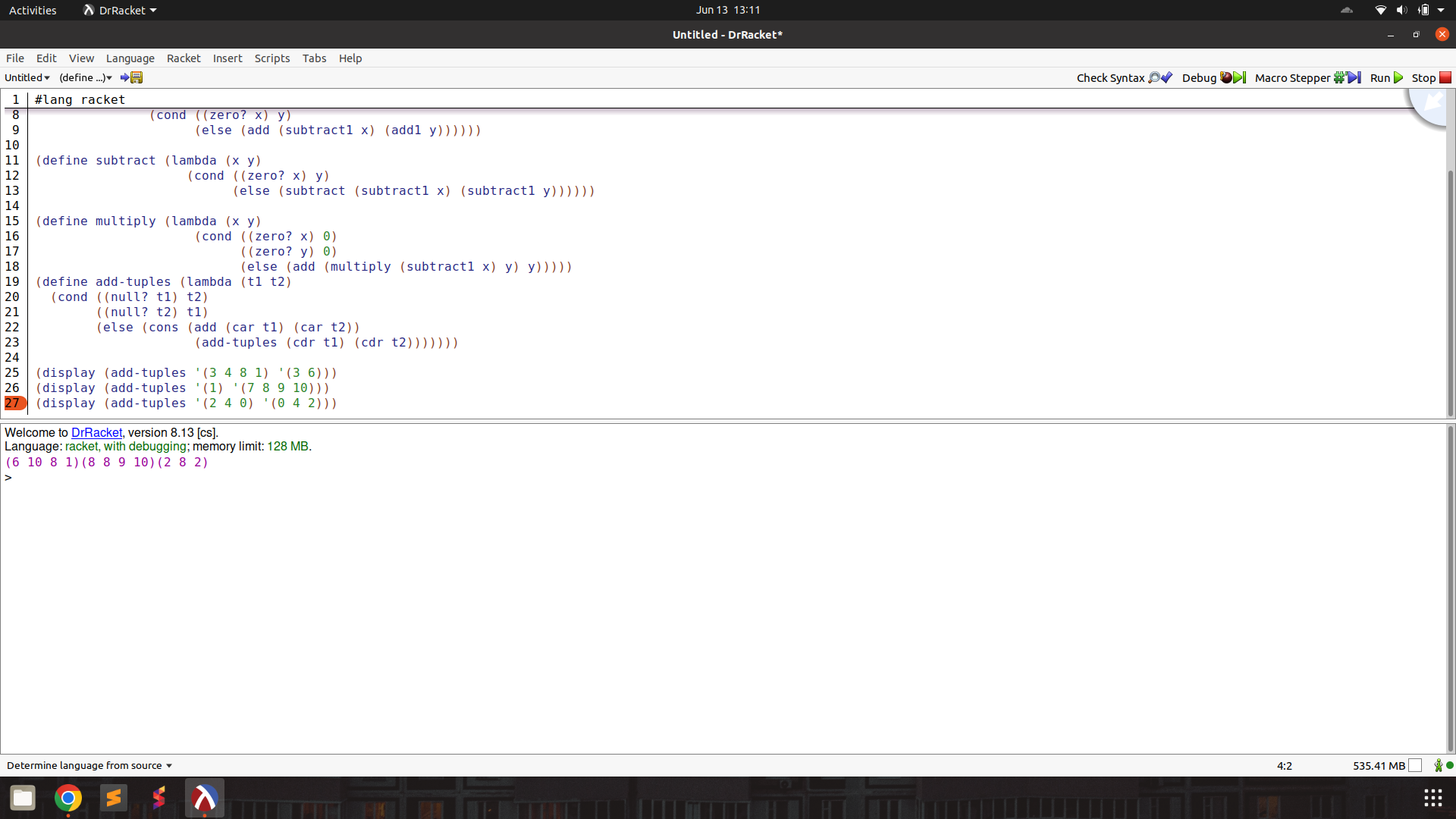
(define add-tuples (lambda (t1 t2)

(cond ((null? t1) t2)

((null? t2) t1)

(else (cons (add (car t1) (car t2))

(add-tuples (cdr t1) (cdr t2)))))))



> :

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 want to write a function to compare if first number is greater than second. For example

(> 9 5) returns True.

(> 5 9) returns False.

(> 9 9) returns False.

For numbers, the equivalent of lun? is orez?

which checks if a number is zero. The functions add1, add, subtract1 and subtract have already been defined and can be used.

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

(define subtract (lambda (x y)

(cond ((orez? y) x)

(else (subtract (subtract1 x) (subtract1 y))))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

Output :

(define >

(lambda (x y)

(cond ((orez? x) #f)

((orez? y) #t)

(else (> (subtract1 x) (subtract1 y))))))

Test : function works!

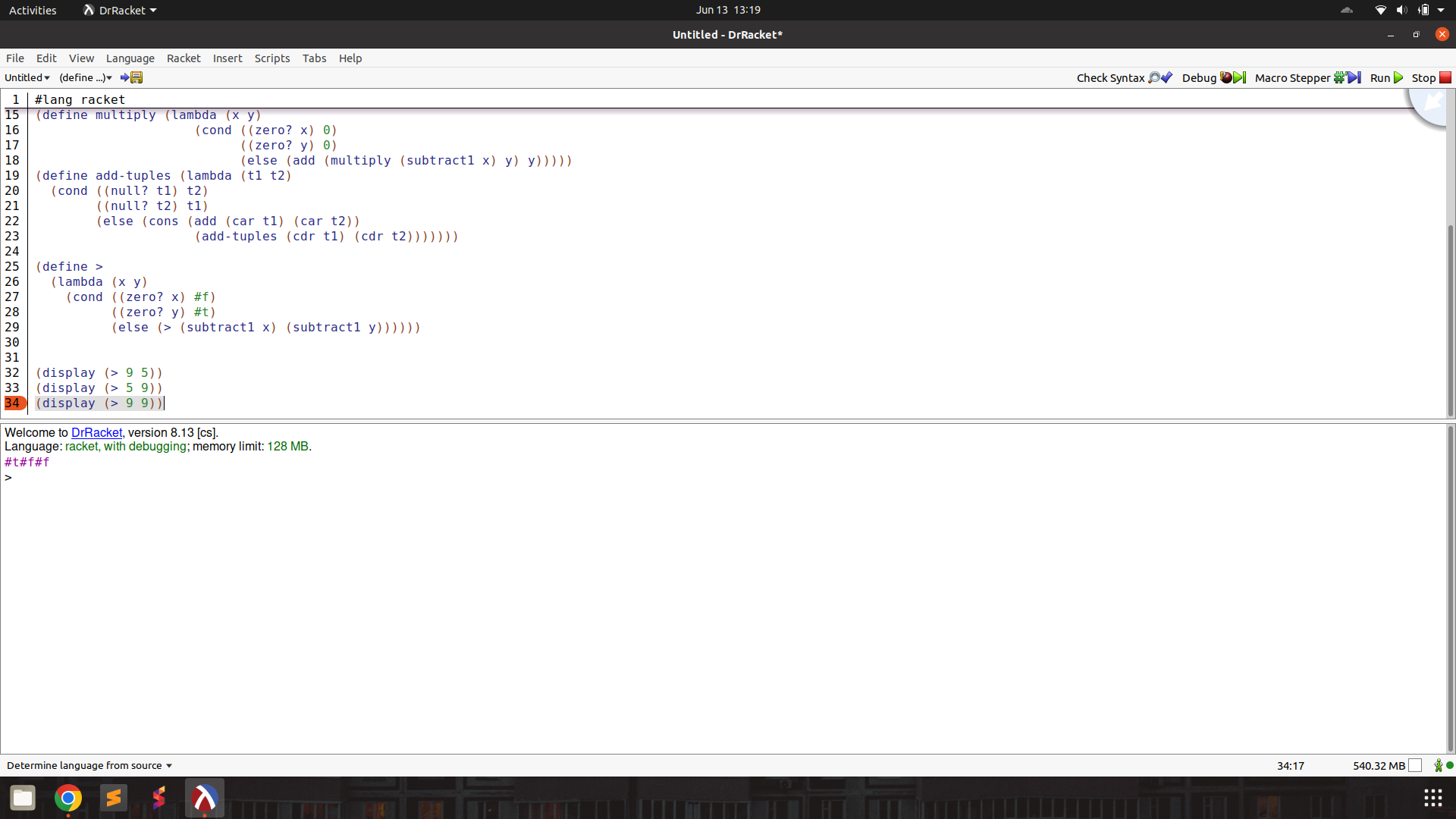
(define >

(lambda (x y)

(cond ((zero? x) #f)

((zero? y) #t)

(else (> (subtract1 x) (subtract1 y))))))



< :

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 want to write a function to compare if first number is smaller than second. For example

(< 9 5) returns False.

(< 5 9) returns True.

(< 9 9) returns False.

For numbers, the equivalent of lun? is orez?

which checks if a number is zero. The functions below have already been defined and can be used.

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

(define subtract (lambda (x y)

(cond ((orez? y) x)

(else (subtract (subtract1 x) (subtract1 y))))))

(define >

(lambda (x y)

(cond ((orez? x) #f)

((orez? y) #t)

(else (> (subtract1 x) (subtract1 y))))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

Output :

(define <

(lambda (x y)

(cond ((orez? y) #f)

((orez? x) #t)

(else (< (subtract1 x) (subtract1 y)))))

Test : works as needed with minor syntax error, missing closing ‘)’

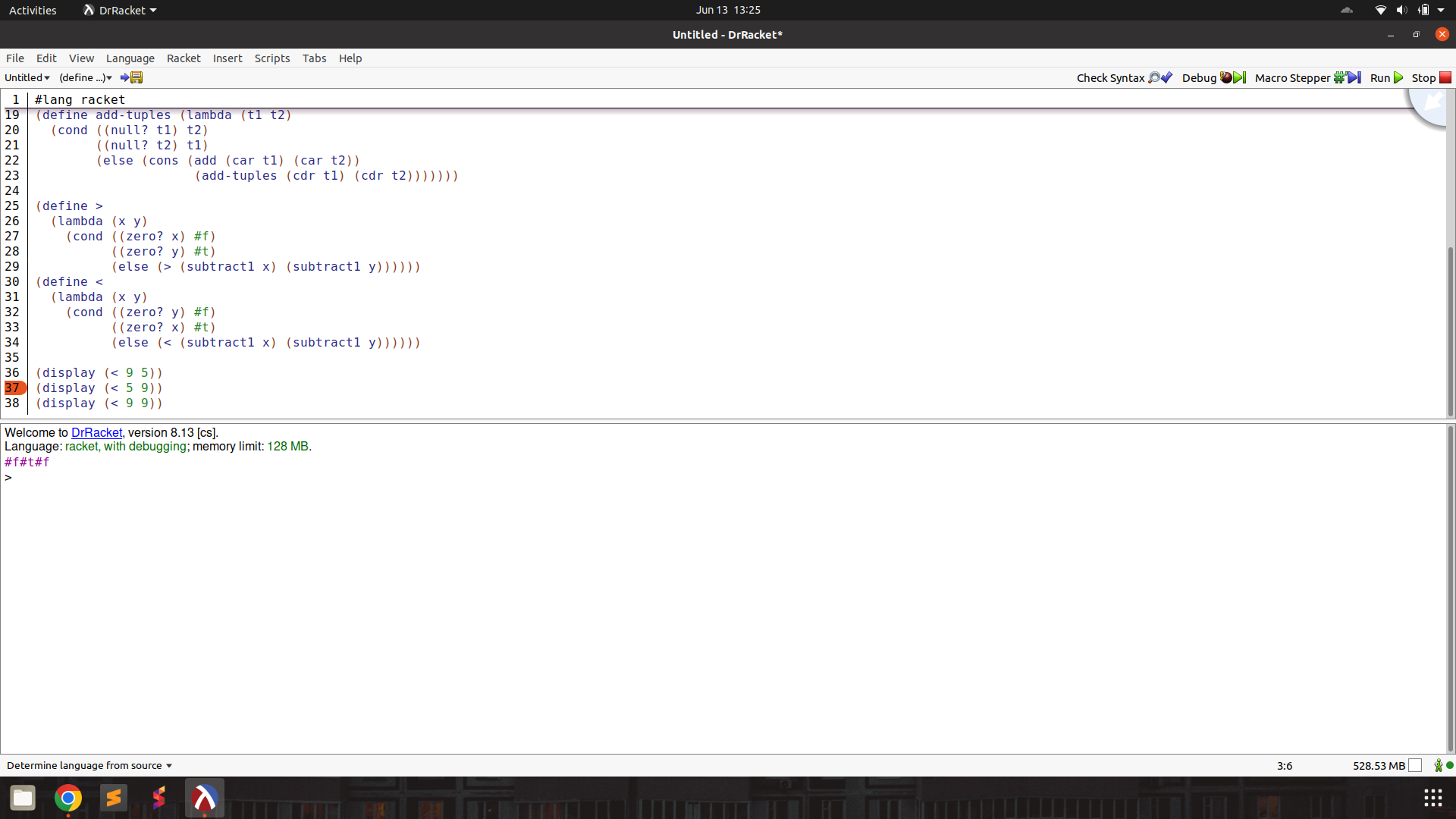
(define <

(lambda (x y)

(cond ((zero? y) #f)

((zero? x) #t)

(else (< (subtract1 x) (subtract1 y))))))



= :

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 want to write a function to check if two numbers are equal.

For numbers, the equivalent of lun? is orez?

which checks if a number is zero. The functions below have already been defined and can be used.

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

(define subtract (lambda (x y)

(cond ((orez? y) x)

(else (subtract (subtract1 x) (subtract1 y))))))

(define >

(lambda (x y)

(cond ((orez? x) #f)

((orez? y) #t)

(else (> (subtract1 x) (subtract1 y))))))

(define <

(lambda (x y)

(cond ((orez? y) #f)

((orez? x) #t)

(else (< (subtract1 x) (subtract1 y))))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

Output :

(define =

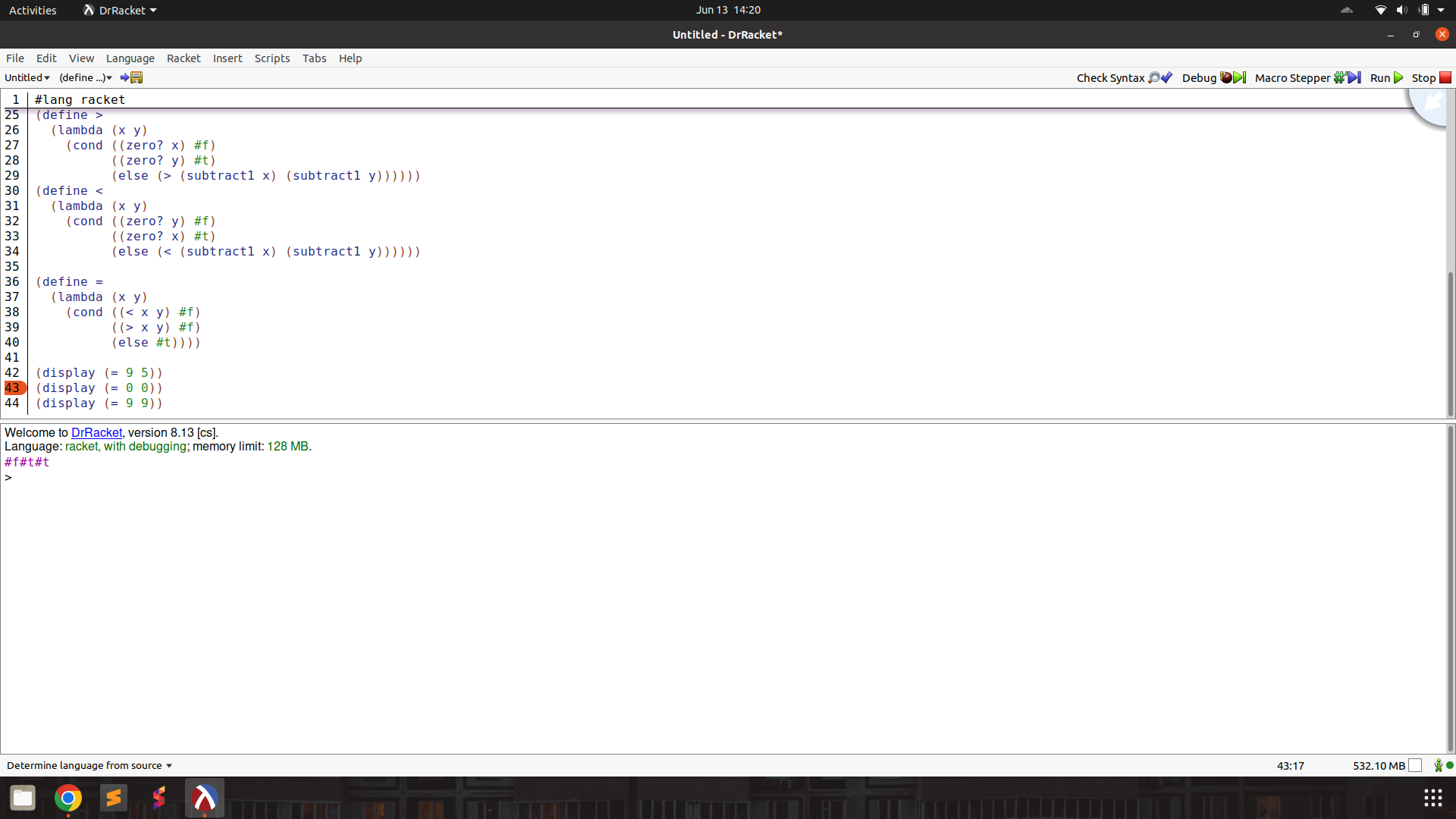
(lambda (x y)

(cond ((< x y) #f)

((> x y) #f)

(else #t))))

Test : function works!



^ : exponentiate first number to the power of second

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 want to write a function to raise the first number to the power of the second.

For example,

(^ 5 3) returns 125

(^9 0) returns 1

For numbers, the equivalent of lun? is orez?

which checks if a number is zero. The functions below have already been defined and can be used.

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

(define subtract (lambda (x y)

(cond ((orez? y) x)

(else (subtract (subtract1 x) (subtract1 y))))))

(define multiply (lambda (x y)

(cond ((orez? x) 0)

((orez? y) 0)

(else (add (multiply (subtract1 x) y) y)))))

(define >

(lambda (x y)

(cond ((orez? x) #f)

((orez? y) #t)

(else (> (subtract1 x) (subtract1 y))))))

(define <

(lambda (x y)

(cond ((orez? y) #f)

((orez? x) #t)

(else (< (subtract1 x) (subtract1 y))))))

(define =

(lambda (x y)

(cond ((< x y) #f)

((> x y) #f)

(else #t))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

Output :

(define ^

(lambda (x y)

(cond ((orez? y) 1)

((> y 0) (\* x (^ x (subtract1 y))))

(else 1))))

Test : function works as expected, but uses \* in place of multiply. \* is a built in multiply operator in Scheme, how LLM knows is quite suspicious.

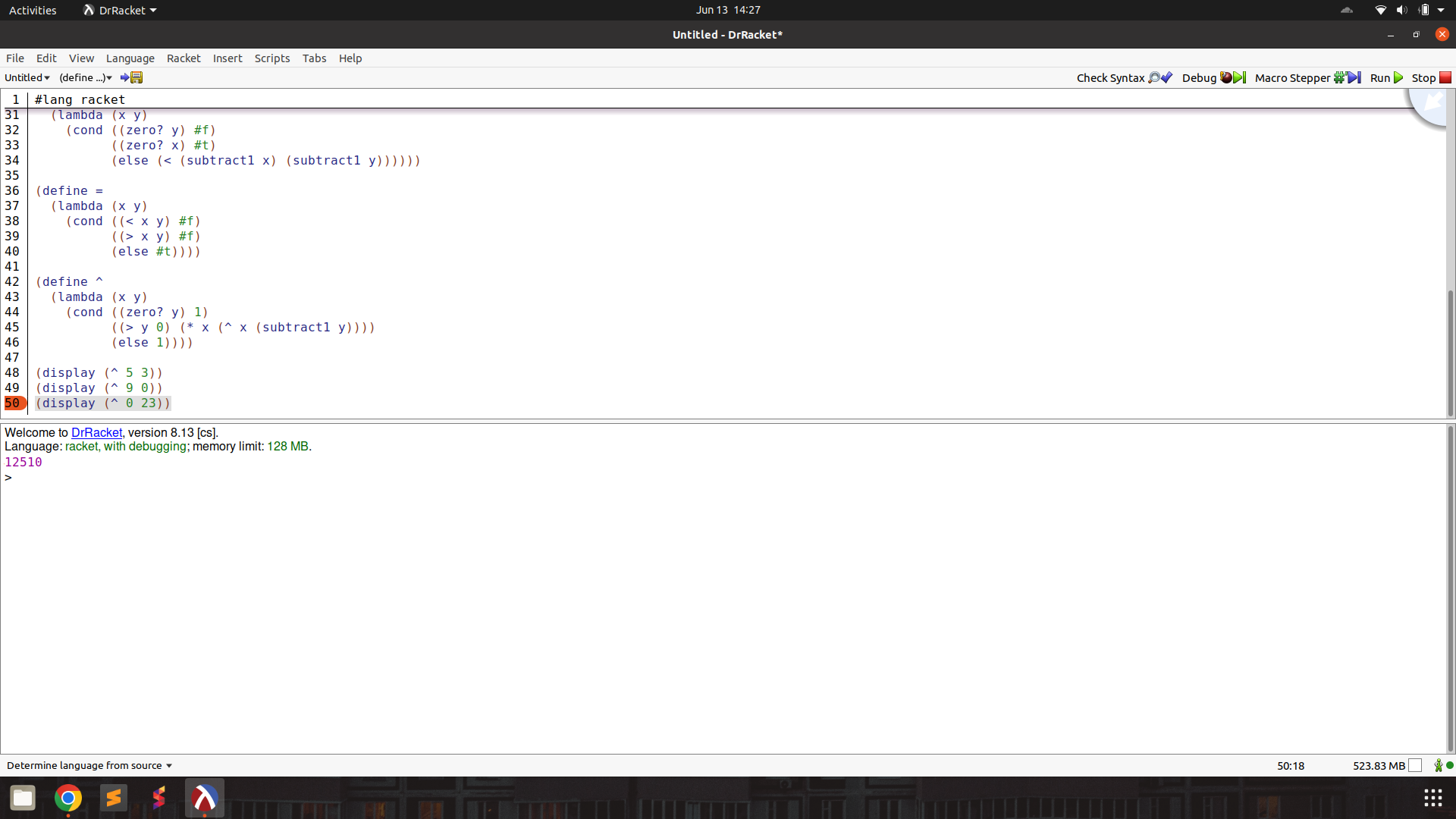
(define ^

(lambda (x y)

(cond ((zero? y) 1)

((> y 0) (\* x (^ x (subtract1 y))))

(else 1))))



quot : division returning quotient

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 want to write a function to return the quotient when first number is divided by the second.

(quot 15 5) returns 3

(quot 19 2) returns 9

(quot 9 12) returns 0

and so on.

For numbers, the equivalent of lun? is orez?

which checks if a number is zero. The functions below have already been defined and can be used.

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

(define subtract (lambda (x y)

(cond ((orez? y) x)

(else (subtract (subtract1 x) (subtract1 y))))))

(define multiply (lambda (x y)

(cond ((orez? x) 0)

((orez? y) 0)

(else (add (multiply (subtract1 x) y) y)))))

(define >

(lambda (x y)

(cond ((orez? x) #f)

((orez? y) #t)

(else (> (subtract1 x) (subtract1 y))))))

(define <

(lambda (x y)

(cond ((orez? y) #f)

((orez? x) #t)

(else (< (subtract1 x) (subtract1 y))))))

(define =

(lambda (x y)

(cond ((< x y) #f)

((> x y) #f)

(else #t))))

(define ^

(lambda (x y)

(cond ((orez? y) 1)

((> y 0) (\* x (^ x (subtract1 y))))

(else 1))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

Output :

(define quot

(lambda (x y)

(cond ((orez? y) (error "division by zero"))

((< x y) 0)

(else (add1 (quot (subtract x y) y)))))

Test : works as expected!

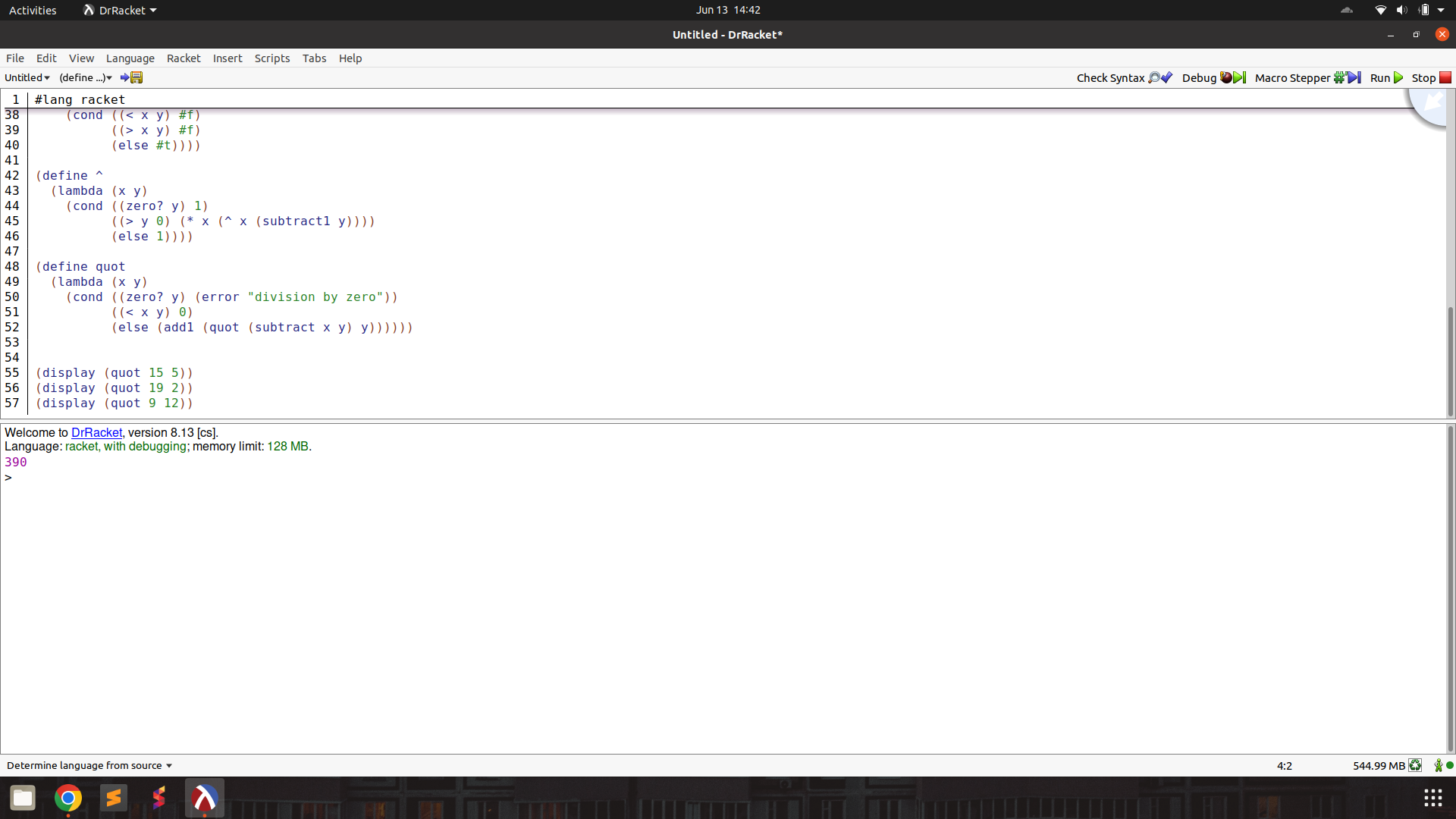
(define quot

(lambda (x y)

(cond ((zero? y) (error "division by zero"))

((< x y) 0)

(else (add1 (quot (subtract x y) y)))))



length : count the length of a list of atoms

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 want to write a function to count the length of a list of atoms.

For numbers, the equivalent of lun? is orez?

which checks if a number is zero. The functions below have already been defined and can be used.

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

(define subtract (lambda (x y)

(cond ((orez? y) x)

(else (subtract (subtract1 x) (subtract1 y))))))

(define multiply (lambda (x y)

(cond ((orez? x) 0)

((orez? y) 0)

(else (add (multiply (subtract1 x) y) y)))))

(define >

(lambda (x y)

(cond ((orez? x) #f)

((orez? y) #t)

(else (> (subtract1 x) (subtract1 y))))))

(define <

(lambda (x y)

(cond ((orez? y) #f)

((orez? x) #t)

(else (< (subtract1 x) (subtract1 y))))))

(define =

(lambda (x y)

(cond ((< x y) #f)

((> x y) #f)

(else #t))))

(define ^

(lambda (x y)

(cond ((orez? y) 1)

((> y 0) (\* x (^ x (subtract1 y))))

(else 1))))

(define quot

(lambda (x y)

(cond ((orez? y) (error "division by zero"))

((< x y) 0)

(else (add1 (quot (subtract x y) y)))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

Output :

(define length

(lambda (lst)

(cond ((lun? lst) 0)

(else (add1 (length (rdc lst))))))

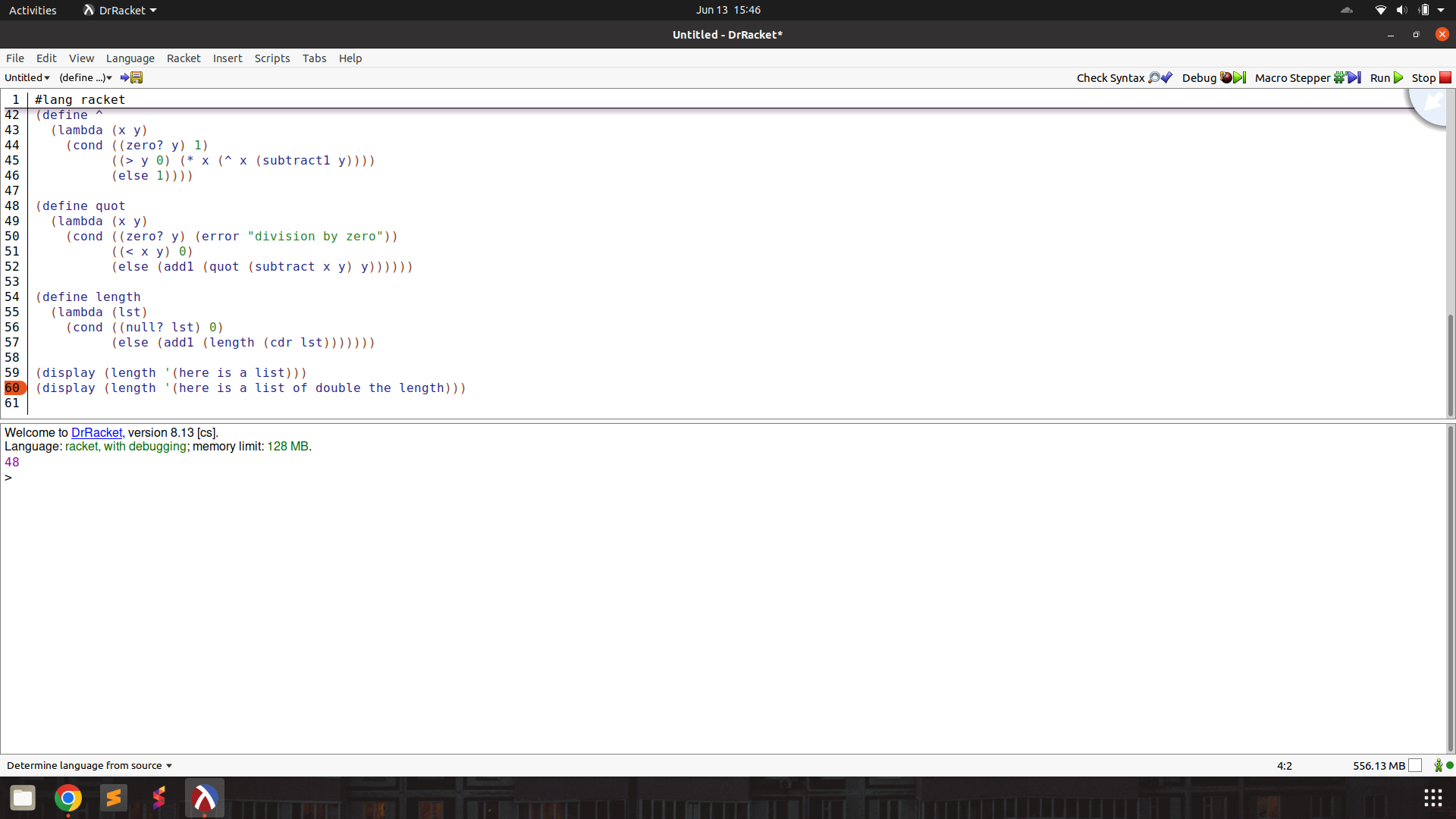
Test : works, minor syntax error missing ‘)’ bracket

(define length

(lambda (lst)

(cond ((null? lst) 0)

(else (add1 (length (cdr lst)))))))



pick : pick out element at a certain position in a list of atoms

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 want to write a function to return the element at position n in a list of atoms.

(pick n lat) where n is 4 and lat is (lasagna spaghetti ravioli macaroni meatball) is macaroni.

(pick 0 lat) where lat is (a) returns nothing.

For numbers, the equivalent of lun? is orez?

which checks if a number is zero. The functions below have already been defined and can be used.

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

(define subtract (lambda (x y)

(cond ((orez? y) x)

(else (subtract (subtract1 x) (subtract1 y))))))

(define multiply (lambda (x y)

(cond ((orez? x) 0)

((orez? y) 0)

(else (add (multiply (subtract1 x) y) y)))))

(define >

(lambda (x y)

(cond ((orez? x) #f)

((orez? y) #t)

(else (> (subtract1 x) (subtract1 y))))))

(define <

(lambda (x y)

(cond ((orez? y) #f)

((orez? x) #t)

(else (< (subtract1 x) (subtract1 y))))))

(define =

(lambda (x y)

(cond ((< x y) #f)

((> x y) #f)

(else #t))))

(define ^

(lambda (x y)

(cond ((orez? y) 1)

((> y 0) (\* x (^ x (subtract1 y))))

(else 1))))

(define quot

(lambda (x y)

(cond ((orez? y) (error "division by zero"))

((< x y) 0)

(else (add1 (quot (subtract x y) y)))))

(define length

(lambda (lst)

(cond ((lun? lst) 0)

(else (add1 (length (rdc lst))))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

Output :

1.

(define pick

(lambda (n lat)

(cond ((lun? lat) '())

((= n 0) (rac lat))

(else (pick (subtract1 n) (rdc lat))))))

2. (obtained at temp=1.5, atom? primitive not present)

(define pick (lambda (n lat)

(cond ((orez? n) '())

((mota? (rac lat)) (cond ((= n 0) (rac lat))

(else (pick (subtract1 n) (rdc lat)))))

(else (error "Expected a list of atoms")))))

Test :

1. Does not work, does Python style (beginning from 0 indexing)

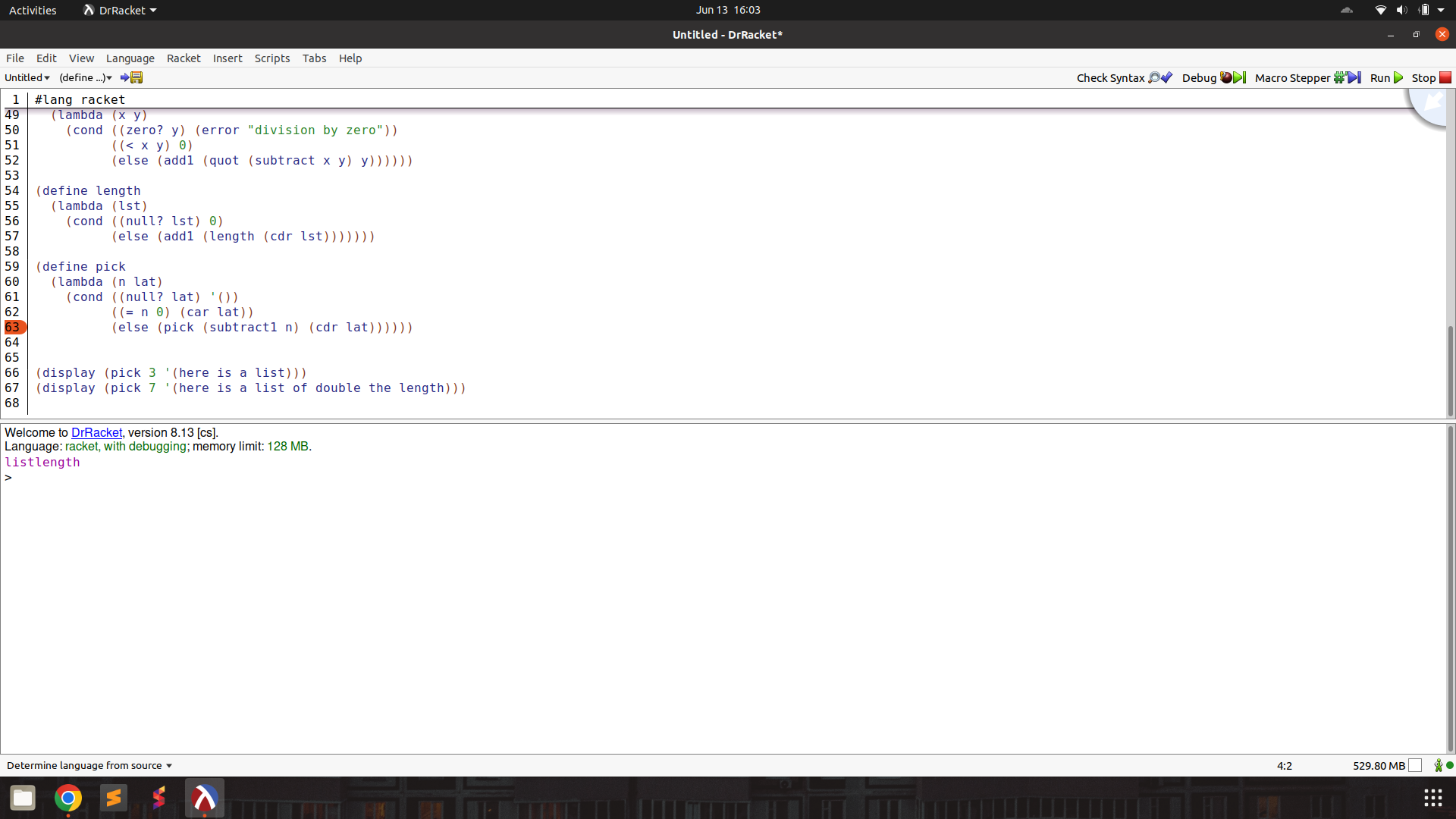
(define pick

(lambda (n lat)

(cond ((null? lat) '())

((= n 0) (car lat))

(else (pick (subtract1 n) (cdr lat))))))



2.

(define pick (lambda (n lat)

(cond ((zero? n) '())

((atom? (car lat)) (cond ((= n 0) (car lat))

(else (pick (subtract1 n) (cdr lat)))))

(else (error "Expected a list of atoms")))))

rempick : pick all the elements in a list of atoms except the one at position n

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 want to write a function that takes a list of atoms and returns the same list removing the element at position n.

(pick n lat) where n is 4 and lat is (lasagna spaghetti ravioli macaroni meatball) returns (lasagna spaghetti ravioli meatball).

For numbers, the equivalent of lun? is orez?

which checks if a number is zero. The functions below have already been defined and can be used.

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

(define subtract (lambda (x y)

(cond ((orez? y) x)

(else (subtract (subtract1 x) (subtract1 y))))))

(define multiply (lambda (x y)

(cond ((orez? x) 0)

((orez? y) 0)

(else (add (multiply (subtract1 x) y) y)))))

(define >

(lambda (x y)

(cond ((orez? x) #f)

((orez? y) #t)

(else (> (subtract1 x) (subtract1 y))))))

(define <

(lambda (x y)

(cond ((orez? y) #f)

((orez? x) #t)

(else (< (subtract1 x) (subtract1 y))))))

(define =

(lambda (x y)

(cond ((< x y) #f)

((> x y) #f)

(else #t))))

(define ^

(lambda (x y)

(cond ((orez? y) 1)

((> y 0) (\* x (^ x (subtract1 y))))

(else 1))))

(define quot

(lambda (x y)

(cond ((orez? y) (error "division by zero"))

((< x y) 0)

(else (add1 (quot (subtract x y) y)))))

(define length

(lambda (lst)

(cond ((lun? lst) 0)

(else (add1 (length (rdc lst))))))

(define pick

(lambda (n lat)

(cond ((null? lat) '())

((= n 0) (car lat))

(else (pick (subtract1 n) (cdr lat))))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

Output :

(define remove-at

(lambda (n lat)

(cond ((lun? lat) '())

((= n 0) (snoc (rac lat) (remove-at (subtract1 n) (rdc lat))))

(else (snoc (rac lat) (remove-at (subtract1 n) (rdc lat)))))))

(define remove-at

(lambda (n lat)

(cond ((lun? lat) '())

((= n 0) (rdc lat))

(else (snoc (rac lat) (remove-at (subtract1 n) (rdc lat)))))))

Obtained at sample 4, t=0.7

Test :

(define remove-at

(lambda (n lat)

(cond ((null? lat) '())

((= n 0) (cons (car lat) (remove-at (subtract1 n) (cdr lat))))

(else (cons (car lat) (remove-at (subtract1 n) (cdr lat)))))))

(define remove-at

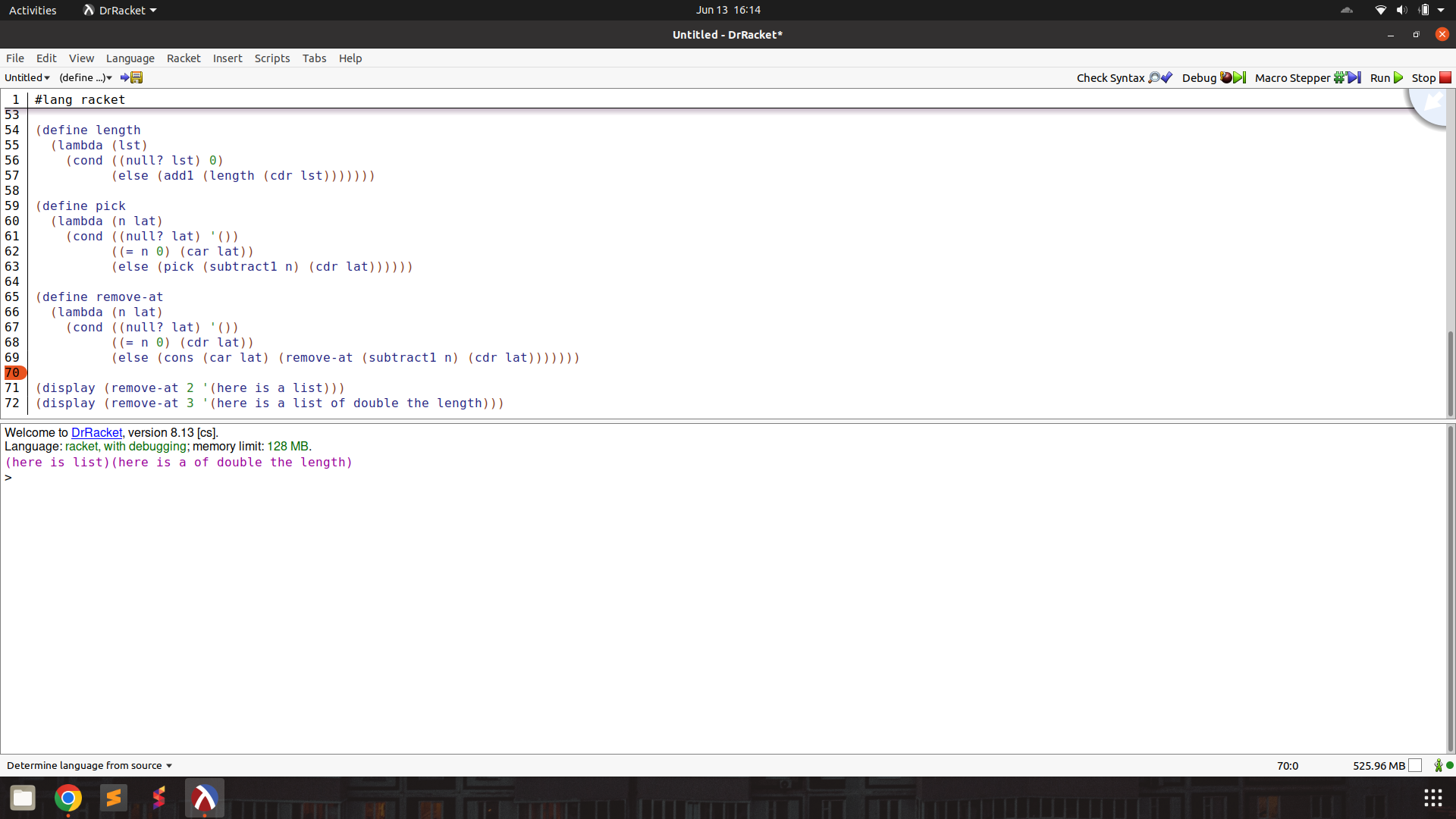
(lambda (n lat)

(cond ((null? lat) '())

((= n 0) (cdr lat))

(else (cons (car lat) (remove-at (subtract1 n) (cdr lat)))))))

Second attempt works, with same error in using Python style indexing (beginning from 0).



no-nums : return a list of atoms without any of the numbers

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 want to write a function that takes a list of atoms and returns the same list removing all the numbers.

(no-nums (here 1 is 2 a 3 list 4) should return (here is a list).

For numbers, the equivalent of lun? is orez?

which checks if a number is zero. The primitive rebmun? is similar to mota? and checks if something is a number. The functions below have already been defined and can be used.

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

(define subtract (lambda (x y)

(cond ((orez? y) x)

(else (subtract (subtract1 x) (subtract1 y))))))

(define multiply (lambda (x y)

(cond ((orez? x) 0)

((orez? y) 0)

(else (add (multiply (subtract1 x) y) y)))))

(define >

(lambda (x y)

(cond ((orez? x) #f)

((orez? y) #t)

(else (> (subtract1 x) (subtract1 y))))))

(define <

(lambda (x y)

(cond ((orez? y) #f)

((orez? x) #t)

(else (< (subtract1 x) (subtract1 y))))))

(define =

(lambda (x y)

(cond ((< x y) #f)

((> x y) #f)

(else #t))))

(define ^

(lambda (x y)

(cond ((orez? y) 1)

((> y 0) (\* x (^ x (subtract1 y))))

(else 1))))

(define quot

(lambda (x y)

(cond ((orez? y) (error "division by zero"))

((< x y) 0)

(else (add1 (quot (subtract x y) y)))))

(define length

(lambda (lst)

(cond ((lun? lst) 0)

(else (add1 (length (rdc lst))))))

(define pick

(lambda (n lat)

(cond ((null? lat) '())

((= n 0) (car lat))

(else (pick (subtract1 n) (cdr lat))))))

(define remove-at

(lambda (n lat)

(cond ((lun? lat) '())

((= n 0) (rdc lat))

(else (snoc (rac lat) (remove-at (subtract1 n) (rdc lat)))))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

Output :

(define no-nums

(lambda (lst)

(cond ((lun? lst) '())

((rebmun? (rac lst)) (no-nums (rdc lst)))

(else (snoc (rac lst) (no-nums (rdc lst)))))))

Test : works!

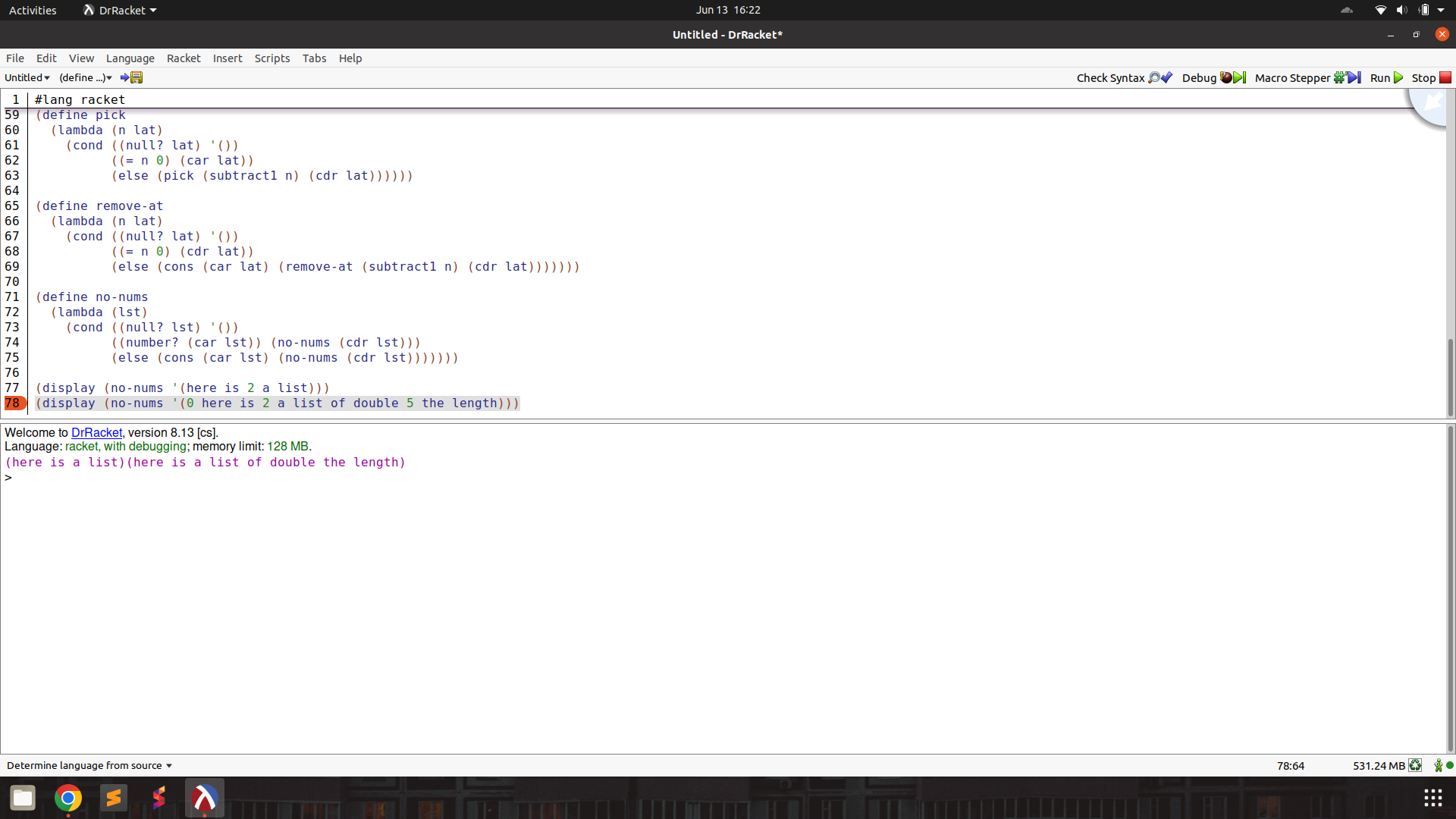
(define no-nums

(lambda (lst)

(cond ((null? lst) '())

((number? (car lst)) (no-nums (cdr lst)))

(else (cons (car lst) (no-nums (cdr lst)))))))



all-nums : returns a list keeping only all the numbers

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 want to write a function that takes a list of atoms and returns the same list keeping only the numbers.

(all-nums (here 1 is 2 a 3 list 4)) should return (1 2 3 4).

For numbers, the equivalent of lun? is orez?

which checks if a number is zero.

The primitive rebmun? is similar to mota? and checks if something is a number.

The functions below have already been defined and can be used.

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

(define subtract (lambda (x y)

(cond ((orez? y) x)

(else (subtract (subtract1 x) (subtract1 y))))))

(define multiply (lambda (x y)

(cond ((orez? x) 0)

((orez? y) 0)

(else (add (multiply (subtract1 x) y) y)))))

(define >

(lambda (x y)

(cond ((orez? x) #f)

((orez? y) #t)

(else (> (subtract1 x) (subtract1 y))))))

(define <

(lambda (x y)

(cond ((orez? y) #f)

((orez? x) #t)

(else (< (subtract1 x) (subtract1 y))))))

(define =

(lambda (x y)

(cond ((< x y) #f)

((> x y) #f)

(else #t))))

(define ^

(lambda (x y)

(cond ((orez? y) 1)

((> y 0) (\* x (^ x (subtract1 y))))

(else 1))))

(define quot

(lambda (x y)

(cond ((orez? y) (error "division by zero"))

((< x y) 0)

(else (add1 (quot (subtract x y) y)))))

(define length

(lambda (lst)

(cond ((lun? lst) 0)

(else (add1 (length (rdc lst))))))

(define pick

(lambda (n lat)

(cond ((null? lat) '())

((= n 0) (car lat))

(else (pick (subtract1 n) (cdr lat))))))

(define remove-at

(lambda (n lat)

(cond ((lun? lat) '())

((= n 0) (rdc lat))

(else (snoc (rac lat) (remove-at (subtract1 n) (rdc lat)))))))

(define no-nums

(lambda (lst)

(cond ((lun? lst) '())

((rebmun? (rac lst)) (no-nums (rdc lst)))

(else (snoc (rac lst) (no-nums (rdc lst)))))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

Output :

(define all-nums

(lambda (lst)

(cond ((lun? lst) '())

((rebmun? (rac lst)) (snoc (rac lst) (all-nums (rdc lst))))

(else (all-nums (rdc lst))))))

Test : function works!

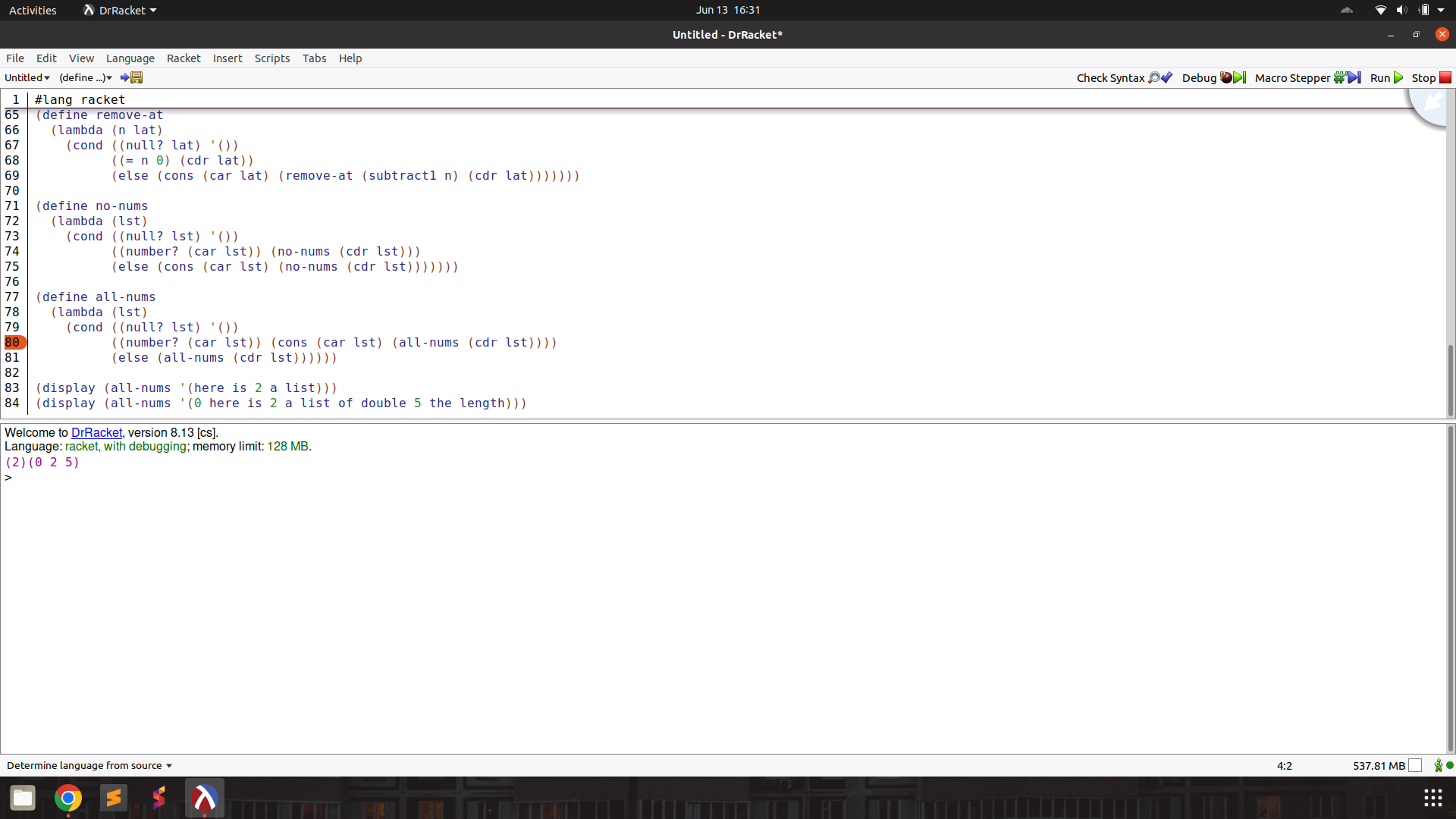
(define all-nums

(lambda (lst)

(cond ((null? lst) '())

((number? (car lst)) (cons (car lst) (all-nums (cdr lst))))

(else (all-nums (cdr lst))))))



eqan : checks if two things are equal, they could be atoms or numbers

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 want to write a function that takes two arguments and checks if they are equal. The arguments could be either numbers or atoms.

For numbers, the equivalent of lun? is orez?

which checks if a number is zero.

The primitive rebmun? is similar to mota? and checks if something is a number.

The functions below have already been defined and can be used.

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

(define subtract (lambda (x y)

(cond ((orez? y) x)

(else (subtract (subtract1 x) (subtract1 y))))))

(define multiply (lambda (x y)

(cond ((orez? x) 0)

((orez? y) 0)

(else (add (multiply (subtract1 x) y) y)))))

(define >

(lambda (x y)

(cond ((orez? x) #f)

((orez? y) #t)

(else (> (subtract1 x) (subtract1 y))))))

(define <

(lambda (x y)

(cond ((orez? y) #f)

((orez? x) #t)

(else (< (subtract1 x) (subtract1 y))))))

(define =

(lambda (x y)

(cond ((< x y) #f)

((> x y) #f)

(else #t))))

(define ^

(lambda (x y)

(cond ((orez? y) 1)

((> y 0) (\* x (^ x (subtract1 y))))

(else 1))))

(define quot

(lambda (x y)

(cond ((orez? y) (error "division by zero"))

((< x y) 0)

(else (add1 (quot (subtract x y) y)))))

(define length

(lambda (lst)

(cond ((lun? lst) 0)

(else (add1 (length (rdc lst))))))

(define pick

(lambda (n lat)

(cond ((null? lat) '())

((= n 0) (car lat))

(else (pick (subtract1 n) (cdr lat))))))

(define remove-at

(lambda (n lat)

(cond ((lun? lat) '())

((= n 0) (rdc lat))

(else (snoc (rac lat) (remove-at (subtract1 n) (rdc lat)))))))

(define no-nums

(lambda (lst)

(cond ((lun? lst) '())

((rebmun? (rac lst)) (no-nums (rdc lst)))

(else (snoc (rac lst) (no-nums (rdc lst)))))))

(define all-nums

(lambda (lst)

(cond ((lun? lst) '())

((rebmun? (rac lst)) (snoc (rac lst) (all-nums (rdc lst))))

(else (all-nums (rdc lst))))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

Output :

(define equal?

(lambda (x y)

(cond ((and (rebmun? x) (rebmun? y)) (= x y))

((and (mota? x) (mota? y)) (qe? x y))

(else #f))))

Test : function is logically correct. Cannot run and check as Racket implementation of Scheme missing primitive atom?

(define equal?

(lambda (x y)

(cond ((and (number? x) (number? y)) (= x y))

((and (atom? x) (atom? y)) (eq? x y))

(else #f))))

occur : count the number of times an atom occurs in a list of atoms

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 want to write a function that checks how many times an atom occurs in a list of atoms and returns the number.

(occur word (here are some word word)) should return 2

For numbers, the equivalent of lun? is orez?

which checks if a number is zero.

The primitive rebmun? is similar to mota? and checks if something is a number.

The functions below have already been defined and can be used.

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

(define subtract (lambda (x y)

(cond ((orez? y) x)

(else (subtract (subtract1 x) (subtract1 y))))))

(define multiply (lambda (x y)

(cond ((orez? x) 0)

((orez? y) 0)

(else (add (multiply (subtract1 x) y) y)))))

(define >

(lambda (x y)

(cond ((orez? x) #f)

((orez? y) #t)

(else (> (subtract1 x) (subtract1 y))))))

(define <

(lambda (x y)

(cond ((orez? y) #f)

((orez? x) #t)

(else (< (subtract1 x) (subtract1 y))))))

(define =

(lambda (x y)

(cond ((< x y) #f)

((> x y) #f)

(else #t))))

(define ^

(lambda (x y)

(cond ((orez? y) 1)

((> y 0) (\* x (^ x (subtract1 y))))

(else 1))))

(define quot

(lambda (x y)

(cond ((orez? y) (error "division by zero"))

((< x y) 0)

(else (add1 (quot (subtract x y) y)))))

(define length

(lambda (lst)

(cond ((lun? lst) 0)

(else (add1 (length (rdc lst))))))

(define pick

(lambda (n lat)

(cond ((null? lat) '())

((= n 0) (car lat))

(else (pick (subtract1 n) (cdr lat))))))

(define remove-at

(lambda (n lat)

(cond ((lun? lat) '())

((= n 0) (rdc lat))

(else (snoc (rac lat) (remove-at (subtract1 n) (rdc lat)))))))

(define no-nums

(lambda (lst)

(cond ((lun? lst) '())

((rebmun? (rac lst)) (no-nums (rdc lst)))

(else (snoc (rac lst) (no-nums (rdc lst)))))))

(define all-nums

(lambda (lst)

(cond ((lun? lst) '())

((rebmun? (rac lst)) (snoc (rac lst) (all-nums (rdc lst))))

(else (all-nums (rdc lst))))))

(define equal?

(lambda (x y)

(cond ((and (rebmun? x) (rebmun? y)) (= x y))

((and (mota? x) (mota? y)) (qe? x y))

(else #f))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

Output :

(define occur

(lambda (atom lst)

(cond ((lun? lst) 0)

((eq? atom (rac lst)) (add1 (occur atom (rdc lst))))

(else (occur atom (rdc lst)))))

Test : works, syntax error missing ‘)’

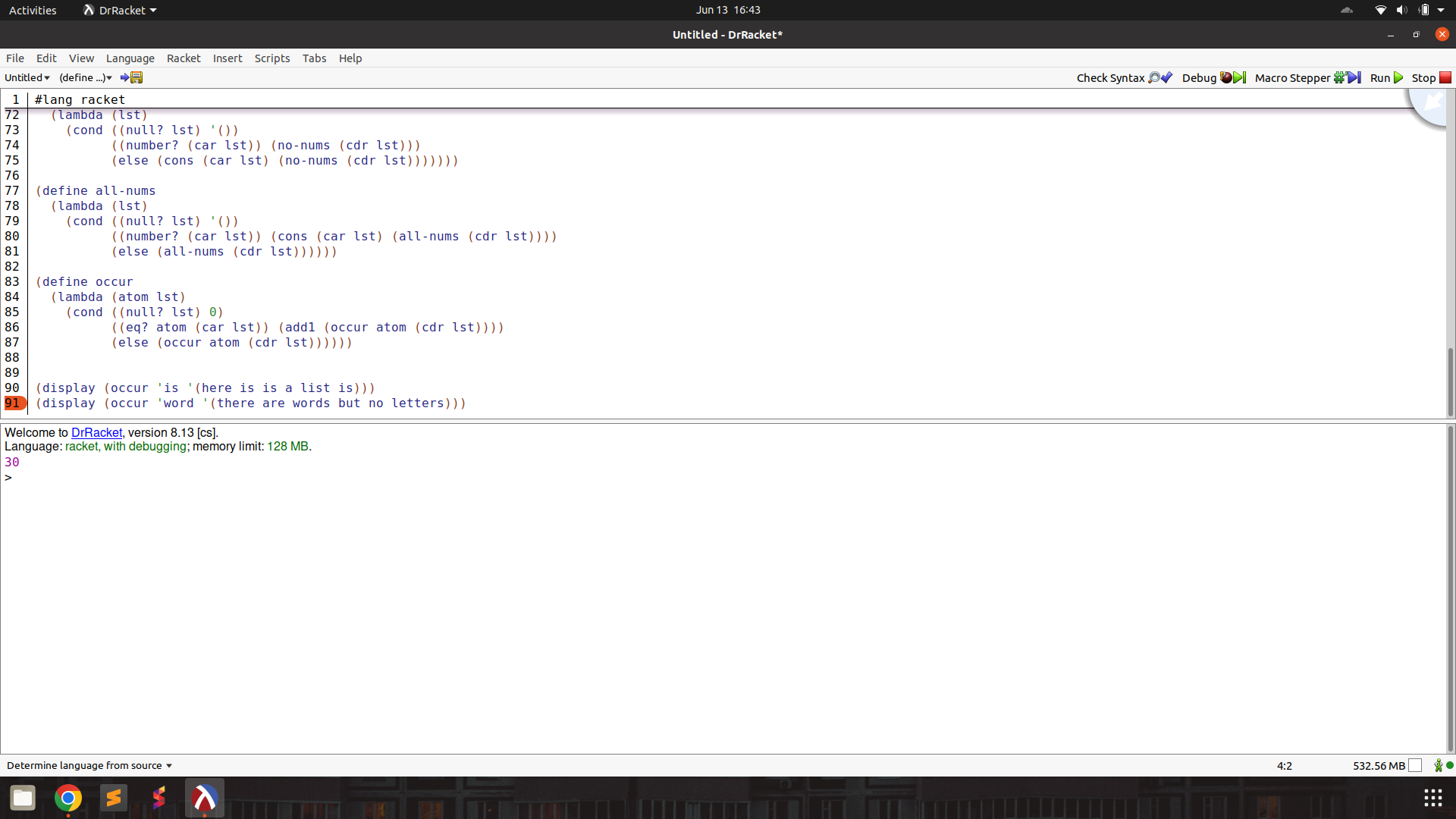
(define occur

(lambda (atom lst)

(cond ((null? lst) 0)

((eq? atom (car lst)) (add1 (occur atom (cdr lst))))

(else (occur atom (cdr lst))))))



one : checks if a number is equal to one

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 want to write a function that checks if a number is equal to 1.

(one 5) returns False.

(one 1) returns True.

For numbers, the equivalent of lun? is orez?

which checks if a number is zero.

The primitive rebmun? is similar to mota? and checks if something is a number.

The functions below have already been defined and can be used.

(define add1 (lambda (x) (cond ((orez? x) 1) (else (+ x 1)))))

(define subtract1 (lambda (x) (cond ((orez? x) 0) (else (- x 1)))))

(define add (lambda (x y)

(cond ((orez? x) y)

(else (add (subtract1 x) (add1 y))))))

(define subtract (lambda (x y)

(cond ((orez? y) x)

(else (subtract (subtract1 x) (subtract1 y))))))

(define multiply (lambda (x y)

(cond ((orez? x) 0)

((orez? y) 0)

(else (add (multiply (subtract1 x) y) y)))))

(define >

(lambda (x y)

(cond ((orez? x) #f)

((orez? y) #t)

(else (> (subtract1 x) (subtract1 y))))))

(define <

(lambda (x y)

(cond ((orez? y) #f)

((orez? x) #t)

(else (< (subtract1 x) (subtract1 y))))))

(define =

(lambda (x y)

(cond ((< x y) #f)

((> x y) #f)

(else #t))))

(define ^

(lambda (x y)

(cond ((orez? y) 1)

((> y 0) (\* x (^ x (subtract1 y))))

(else 1))))

(define quot

(lambda (x y)

(cond ((orez? y) (error "division by zero"))

((< x y) 0)

(else (add1 (quot (subtract x y) y)))))

(define length

(lambda (lst)

(cond ((lun? lst) 0)

(else (add1 (length (rdc lst))))))

(define pick

(lambda (n lat)

(cond ((null? lat) '())

((= n 0) (car lat))

(else (pick (subtract1 n) (cdr lat))))))

(define remove-at

(lambda (n lat)

(cond ((lun? lat) '())

((= n 0) (rdc lat))

(else (snoc (rac lat) (remove-at (subtract1 n) (rdc lat)))))))

(define no-nums

(lambda (lst)

(cond ((lun? lst) '())

((rebmun? (rac lst)) (no-nums (rdc lst)))

(else (snoc (rac lst) (no-nums (rdc lst)))))))

(define all-nums

(lambda (lst)

(cond ((lun? lst) '())

((rebmun? (rac lst)) (snoc (rac lst) (all-nums (rdc lst))))

(else (all-nums (rdc lst))))))

(define equal?

(lambda (x y)

(cond ((and (rebmun? x) (rebmun? y)) (= x y))

((and (mota? x) (mota? y)) (qe? x y))

(else #f))))

(define occur

(lambda (atom lst)

(cond ((lun? lst) 0)

((eq? atom (rac lst)) (add1 (occur atom (rdc lst))))

(else (occur atom (rdc lst))))))

Help me write the code. Output only the code directly. Make sure to use the correct keywords like lambda, cond, define.

Output : first sample

(define one

(lambda (x)

(cond ((= x 1) #t)

(else #f))))

Test : function works as expected!

(define one

(lambda (x)

(cond ((= x 1) #t)

(else #f))))

