Comp 524 - Assignment 5: Read-BasicEval-Print in Lisp

Date Assigned: 10/30/2021

Early Completion Date: Tue 11/09/2021

Completion Date: Fri 11/12/2021

The goals of this assignment are to help you understand the internals of a Lisp interpreter and get practice with tree-based recursion in Java. You will extend a Java lisp interpreter skeleton (implemented by Andrew Vitkus) with new functionality. This project implements a rudimentary lisp interpreter using the Model-View-Controller framework. It is purposely incomplete to allow assignments to extend the functionality it offers. **For all new functions you write, you can assume that functions are called correctly.**

The sections labelled Background duplicate some information given in class lectures and can be skimmed if you understood these lectures well. It ties lecture concepts with code in the skeleton using links to GitHub files. It will probably beneficial to follow these links.

Do not reference the lisp skeleton project cloned from Github – this will cause Gradescope tests to fail. Reference the class jar which has the skeleton. If you have cloned the skeleton project, you can use it to look at the source code and you can even link its source to the jar.

We have an initial set of checks written for this assignment. They will be extended and should remain stable by the time the Gradescope autograder is released.

To understand the semantics you need to implement, it might help to use an implemented Lisp interpreter. Here is a guide to installing one: Installing Clisp

Background: Reference Material

- S-expression
- Lists and Conses
- list
- quote
- eval
- Loading Files
- load
- cond
- cons
- car

cdr
atom
Conditionals
lisp logical operators
PropertyChangeListener

Task 0: Create and Test Java Project

Create Project and Reference External (Library) Code

We will refer to the lisp interpreter skeleton written that is included in the Comp524All.jar, as the skeleton or the predefined interpreter.

Write and Test Initial Main Class

To ensure you are correctly referencing the skeleton, create a main class that calls the main.Main of the skeleton, after importing main.Main. Run this main class and input the following to check that it works:

```
5
(+ 4 5)
.
```

Write and Test Initial Class Registry Class

Create a class implementing the interface main.ClassRegistry with stub (empty) implementations for all required methods except getMain(). Use localchecks to check that your class registry is found.

Functionality Overview

The Lisp interpreter skeleton scans and parses each Lisp expression, which can span multiple lines, into an s-expression. It implements some of the predefined Lisp functions such as +. Your tasks are to craft:

1. a recursive implementation of a method that determines if an s-expression is a list.
2. two different implementations for displaying an s-expression, one that works for an arbitrary s-expression and one that is customized for a list.
3. implementations of the following additional built-in functions:
    a. quote
    b. list
    c. eval
    d. load
    e. cond
    f. <, >, <=, >=
    g. **and**, **or**, **not**

S-Expressions and Associated Skeleton Classes

A Lisp interpreter processes a series of expression strings input by the user. Each expression string is individually scanned and parsed into a data structure called an s-expression. The main.lisp.parser.terms package in the skeleton contains the Java interfaces and predefined implementations s-expressions. All s-expressions implement the interface SExpression and inherit from the abstract class AbstractExpression.

An s-expression can be a *composite* s-expression or an *atom* s-expression.

In the skeleton, an atom s-expression can be an int, float, or string literal (e.g. 5, 5.2, "five") or an identifier. An identifier, in turn, can be one of the predefined constants, **T** and **nil**, or an operator/function name.

The interface Atom describes the methods of an Atom. The *getValue()* method returns its value.

A composite s-expression has two slots, Head and Tail, which both point to (atom or composite) s-expressions. Thus, an s-expression is a special case of a binary tree.

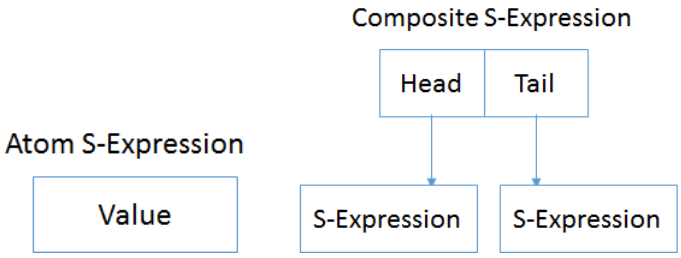

Figure 1: Atom vs Composite S-Expression

Given an s-expression S, S.getHead() and S.getTail() will refer to the s-expressions referenced by the head and tail slots of the s-expression. Also we will call S.getHead() and S.getTail() as *peers* of each other. (H . T) refers to an s-expression whose S.getHead() and S.getTail() are H and T, respectively.

The following is an example of an s-expression.

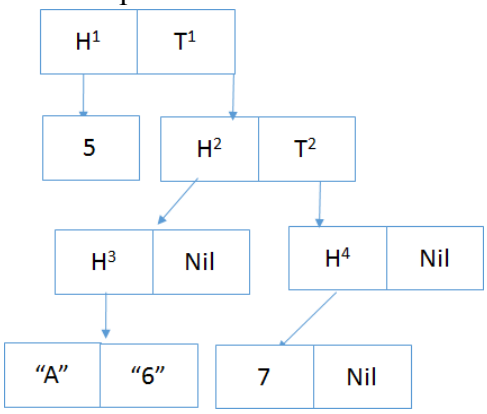

Figure 2: Example of Composite S-Expression

Some of the assignment tasks require you to replace the predefined implementation BasicExpression of a composite s-expression with your own implementation by **inheriting** from the predefined implementation. Its getHead() and getTail() give the head and tail of the expression. Do not modify the predefined implementation directly.

You can check if an s-expression is of a particular type by using the **instanceof** Java operation. Thus:

s **instanceof** Atom

returns true if the s-expression IS-A Atom.

Task 1: isList()

The predefined implementation of a composite s-expression provides a stub method for the method isList(), which returns a boolean. Your implementation of a composite s-expression should override this method from the class main.lisp.parser.terms.BasicExpression by implementing a recursive algorithm with the following semantics:

S.isList: () → boolean

(S is the composite s-expression on which isList() it is invoked).

Let T^1, T^2, ... T^N be the right-most path, from root to leaf, in the tree rooted by S (Figure 3),

return T^N == **nil**.

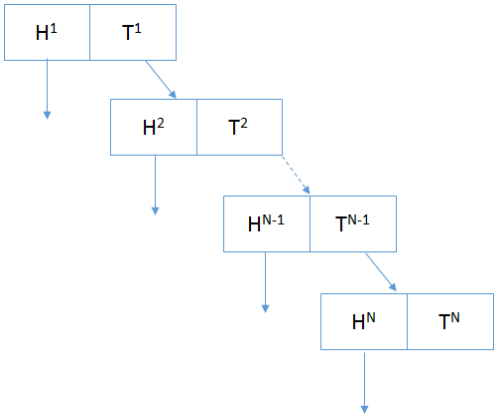

Figure 3: Labelling the Right-Most Path of an S-Expression

That is, an s-expression is a *list* if the right-most path in the tree it roots ends in the atom **nil**. Such an s-expression is essentially a linked list of nodes, ($S^1 = (H^1 . T^1)$ ... $S^N = (H^N T^N)$, where H^i is the value of S^i and T^i is the pointer (link) to S^{i+1}. We will refer to H^i as the ith element of the list. Thus 5, (("A" . "6")), and (7) are the first, second and third elements of the list (5 (("A" . "6")) (7)).

Figure 4 gives some examples of list s-expressions.

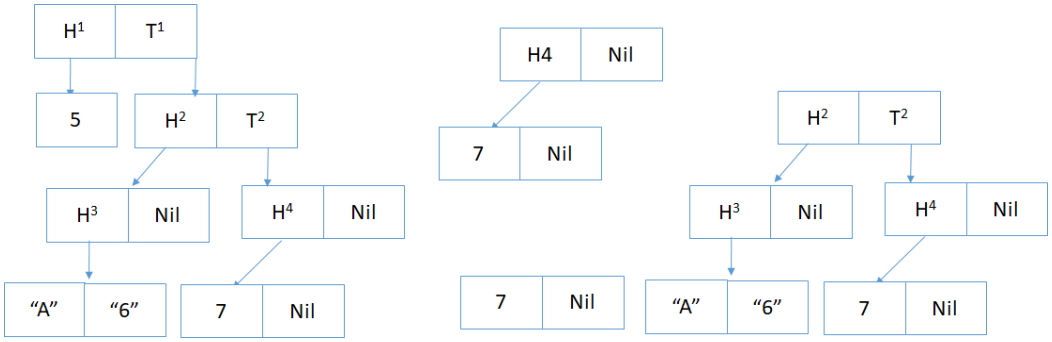

Figure 4: Examples of List Composite S-Expressions

Figure 5 gives examples of composite s-expressions that are not lists, which we will refer to as *tuples*.

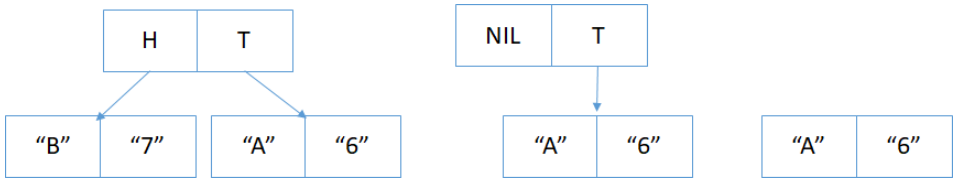

Figure 5: Examples of Non-List Composite S-Expressions

Task 2: Creating the Output Representation of an S-expression

Implement the following public methods in a compound s-expression. Their definition are given using pseudo code.

```
S.toStringAsSExpressionDeep: () → String
        return "(" + S.getHead().toStringAsSExpressionDeep() + " " + "." + " " +
                S.getTail().toStringAsSExpressionDeep() + ")"

S.toStringAsSExpression: () → String
        return "(" + S.getHead().toString() + " " + "." + " " +
                S.getTail().toString() + ")"

S.toStringAsList: () → String
        Let H^i be the i^th member of S
        return "(" + H^1.toString() + " " + H^2.toString() + " " + … H^N.toString() + ")";

S.toString():  () → String
        return S.isList()? S.toStringAsList() :S.toStringAsSExpression().
```

The headers of all of these methods are defined in SExpression. The atom classes provide complete implementations of these methods that simply return the textual representation of the different kinds of atoms. The implementation of these methods in BasicExpression throw exceptions. Your task is to replace BasicExpression with your own implementation of a

compound s-expression that follows these constraints and use the implementations of these methods in atom s-expressions.

The pseudo-code above more or less gives you the Java code you must write. The only non-obvious function is toStringAsList(). Your implementations should use recursion and you should not put a pointer from an s-expression to its peer. In fact, you should not need to add any instance variables in your implementation.

To implement toStringAsList() you can use the **public** helper method defined in the abstract class main.lisp.parser.terms.AbstractSExpression:

toStringAsListHelperPublic () String

Again, the atom classes contain an implementation of this method that returns their textual representation and your task is to provide an implementation of it for a compound s-expression. You will have to cast the interface *SExpression* to the abstract class *AbstractSExpression* to access this method. This method has been made public because some Java implementations do not seem to implement the **protected** access privilege as we understand them. It has not been put in the interface as it is a helper method.

To illustrate the behavior of some of the functions you must implement, toString() invoked on the leftmost list in Figure 4 should return: (5 ((“A” . ”6”)) (7))
toStringAsSExpressionDeep() on the same list should return:

(5 . (((“A” . “6”) . nil) . ((7 . nil) . nil)))

toStringAsSExpression() on the same list should return:

(5 . ((“A” . “6”)) (7))

We will refer to the string returned by S.toString() as the *output representation* of the string. We identify an s-expression by its output representation. Thus, we refer to the leftmost list in Figure 4 as:

(5 ((“A” . ”6”)) (7))

Strictly speaking, **nil** is a list. However, we will leave it ambiguous whether (isList Nil) returns **true** or not. **nil** should print as “Nil” rather than “()”. So **nil** prints like a regular atom not a list. If you define isList (**nil**) as T, then you will have three rather than two cases in toString(). Otherwise, you have two cases.

Task 3: Registering your compound S-expression class

The skeleton parses a multi-line input of an s-expression into an instance of an s-expression object, calls the toString() method on the s-expression, and then calls a predefined eval() function on the expression. The main method in main.Main performs these tasks.

You need to ensure that it creates your s-expression object instead of the predefined s-expression. The factory class main.lisp.parser.terms.ExpressionFactory allows the skeleton to determine which implementation is used. Invoking its setClass() method with the class object representing

your expression class registers your class with it, that is, makes it use your class for an s-expression. The class object for class named C is identified as C.class.

Define your own main class that registers your s-expression with the expression factory and then calls the main() method of the predefined skeleton main class. If you have already created a main class in task 0, add the code to do this registration.

Also in your implementation of the class registry return your expression class object in the appropriate getter. Run local checks again.

Testing Output Representations

Run your main class.

Test your toString() method by inputting some s-expressions involving the Lisp functionality supported by the skeleton. This includes:

(a) predefined int, float and String literals.
(b) arithmetic operations: +, -, *, /
(c) car, cdr, cons, null, atom
(d) =, /=

atom and null return true if their argument is an atomic s-expression and the **nil** value, respectively. = and /= are equals and not equals.

As shown earlier**, e**nter "." to terminate the interpreter.

The interpreter does not support variables, basic functions other than the ones listed above, or declaration of programmer-defined Lisp functions. It will crash if you use them in your input.

Background: Lisp Basic Functions

Like other languages, Lisp supports (a) strings, integers, floats, Booleans and other basic types, and (b) identifiers, called symbols, which can either be variables or named constants. **T** and **nil** are predefined constants denoting the true value and null values, respectively. All other identifiers are variables. Operators such as + and - are also considered identifiers/symbols.

Perhaps the most distinctive feature of Lisp, embodied in its name, is its support for lists, which correspond to Java linked lists in that they have a variable number of elements accessed by following pointers.

The empty list can be entered either as:

()

or

nil

A list is stored as a linked list, and the use of **nil** indicates the linked list for an empty list has only a single slot pointing to nothing.

Lists can be created by executing operations on existing lists such as **nil**. The two-argument cons operation adds its first argument to be added to the front of an existing list denoted by its second argument.

Here are some examples, showing the output representation of lists created by executing cons operations.

(cons 5 nil) → (5)
(cons 6.2 (cons 5 nil)) → (6.2 5)
(cons "Hello" (cons 6.2 (cons 5 nil))) → (“Hello” 6.2 5)
(cons (cons "A" (cons 4 nil)) (cons 6.2 (cons 5 nil))) → (("A" 4) 6.2 5)

As we see above, lists can be nested.

The car operation, when invoked on a non-empty list L, returns the head of the list, that is, the element at the front of L:

(car (cons 5 nil)) → 5
(car (cons 6.2 (cons 5 nil))) → 6.2
(car (cons (cons "A" (cons 4 nil)) (cons 6.2 (cons 5 nil)))) → ("A" 4)

The cdr operation, when invoked on a non-empty list, L, returns the tail of the list, that is, a part of L that does not include its first element:

(cdr (cons 5 nil)) → nil
(cdr (cons 6.2 (cons 5 nil))) → (5)
(cdr (cons (cons "A" (cons 4 nil)) (cons 6.2 (cons 5 nil)))) → (6.2 5)

Lists can be used to represent both data and code. Code consists of function calls and function declarations. Function declarations, in turn, are calls to special functions that create the declarations.

Basic functions are implemented in the language in which the Lisp interpreter is written. Examples are + and -. Other functions may be predefined or programmer-defined. A predefined function is provided by the interpreter and may be implemented in Lisp or its implementation function.

Lists are converted to code by **evaluating** them. Given a list:

$(A^1\ A^2\ ..\ A^N)$

evaluation of the list consists of calling the function named by A^1 with the results of the evaluation of actual parameters $A^2..A^n$. Thus, the list:

(+ 5 3)

is evaluated by calling + with the arguments 5 and 3.

Entering the text:

$(A^1\ A^2\ ..\ A^N)$
asks the Lisp interpreter to first construct the associated list and then evaluate it.

Usually the arguments of a function call are evaluated before passing them to the function. Thus:
(- (+ 5 3) 3)
returns 5.

An exception is the basic function, quote, which simply returns its single argument, unevaluated. Thus, while entering:
(cons 5 nil)
returns
(5)
entering:
(quote (cons 5 nil))
returns:
(cons 5 nil)

A list can be explicitly evaluated by calling the basic single-argument eval function. Thus, calling:
(eval (quote (cons 5 nil)))
returns
(5)
Like arguments of most other functions (except quote and cond), the argument of eval is evaluated before any function-specific processing. The function-specific processing in this case is eval. So the evaluated argument is itself evaluated to give the result. In the example above, the first evaluation gives:

(cons 5 nil)

and the second one gives:

(5)
As we will see later, like lists, variables can also be evaluated, explicitly or implicitly.

Besides lists, Lisp supports a second data structure, called a tuple, which corresponds to a C struct or Java object, in that it has a fixed number of components. It has exactly two elements, the head and tail components, and has the following output representation:
(<Head Component> . <Tail Component>)
The following are examples of tuples:
(“Five” . 5)
((“Five”) . 5)
((“Five” . 5) . (“One” . 1))
As we see above, lists and tuples can be nested in tuples. Similarly, lists can nest tuples:
((“Five” . 5) (“One” . 1) (“Two” . 2))

Note that the textual representation of a list separates elements with spaces whereas the separator " . " is used for tuples.

The car, cdr, and cons operations apply also to tuples.

S = (H . T) can be input as:
    (cons H T)
The example tuples above can be constructed as:
    (cons "Five" 5)
    (cons (cons "Five" nil) 5)
    (cons (cons "Five" 5) (cons "One" 1))
(Clisp will print the last expression using a different format.)

Given S = (H . T)
    (car S) → H
    (cdr S) → T

The fact that car, cdr, and cons apply to both tuples and lists indicates that there is some underlying abstraction uniting them. This abstraction is called an s-expression.

Skeleton Support for Lisp Basic Functions

As mentioned above a call to function F with arguments $A^1 .. A^N$ is represented as a list s-expression of the form:
$L = (F\ A^1 .. A^N)$

As it currently supports only basic functions, the predefined eval() method of a list s-expression invokes the toString() method of F to lookup a registered *evaluator object*, on which it makes the following call:
    eval (args, null)
where args = (cdr L) == $(A^1 .. A^N)$. The eval method of an evaluator object should return an s-expression that represents the result of evaluating $L = (F\ A^1 .. A^N)$.

An evaluator object is an instance of the predefined type, main.lisp.evaluator.Evaluator and must implement the following signature:
```
public SExpression eval(SExpression expr, Environment environment)
```
It is registered by calling the method:
```
public void registerEvaluator(String name, Evaluator evaluator)
```
of the singleton instance of the predefined type main.lisp.evaluator.OperationManager. This instance can be retrieved by calling the static method:
```
public static OperationManager get()
```
of the factory class main.lisp.evaluator.BuiltinOperationManagerSingleton

The class main.lisp.evaluator.BasicOperationRegisterer registers the basic operations registered by the skeleton. Look at its code and the objects it registers to understand how you should implement the required new functions, described below.

Similarly, you will need to write your own class implementing main.lisp.evaluator.OperationRegisterer to register your new evaluators with the interpreter and call its registration method in your main method. This feature will either not be checked in this assignment or will be worth zero points and is added to help you structure your solution.

Task 4: Implement New Basic Operations

As mentioned in the overview, you must implement the following functions as basic operations:

h. quote
i. list
j. eval
k. load
l. cond
m. <, >, <=, >= (relational operators)
n. **and**, **or**, **not (**logical operator**s)**

Look at main.lisp.evaluator.basic.EqEvaluator, main.lisp.evaluator.basic.ProductEvaluator and other predefined evaluators to understand how the familiar relational and Boperations should be implemented by your Java evaluators. Their semantics are given below.

True/False

As in C, true and false are not values in lisp. Instead false is defined as nil and true is any other value. However, there is a standard representation for a generic true value, the atom **T**. Some operators, such as comparisons, will return **T** or **nil** as true or false whereas others such as and and or will return more meaningful values. **T** and **nil** are represented in the interpreter with instances of the TAtom and NilAtom classes respectively.

Here are some examples of true/false values:

True: 1, (+ 1 2), T, (cons 1 2), (list 1 2 3 4)
False: nil, ()

Execution Order

All lisp instructions are processed in order and arguments are processed first to last. Additionally, lisp operations will short-circuit. This means that when an operation has determined its return value, it stops processing any other arguments. This is relevant to operators such as and, or, and cond. In the next assignment we will implement variables and functions so executing extra arguments can result in incorrect state changes.

For example, the following code should result in a being equal to 2, not 4.

```
(setq a 2)
(defun squareA () (setq a (* a a)))
(or T (squareA))
```

List

The evaluator of the list function assumes the s-expression, S passed to it is a list with elements H^1... H^N. (Recall that these are peers of the tails in the rightmost path of the tree representing S). It should return a new list R = (H^1.eval() ... H^N.eval()).

Thus:

```
(list (+ 3 2) (- 7 5))
```

returns the list:

```
(5 2)
```

The evaluator for the list will follow the head and tail pointers in the tree created for S to retrieve its elements, call the eval() function on each of its elements, and create a new list containing the results of eval(). The list is constructed by iteratively calling the ExpressionFactory.newInstance(SExpression, SExpression) method with the head and tail of each s-expression in the list, noting that SExpressions are immutable and thus cannot be changed after creation. Therefore, the order in which the list is created is important since each s-expression must know its head and tail before creation.

Use the predefined factory class to instantiate all s-expression objects. Thus, your evaluators should not make any assumptions about the class implementing an s-expression.

Quote

As mentioned in the background section above, the quote operation returns its argument unevaluated. Thus:

```
(quote (cons “one” 1))
```

returns the s-expression.

```
(cons “one” 1)
```

Eval

As also mentioned in the background section above, the eval operation returns the result of calling eval() on its *evaluated* argument. Thus:

```
(eval (quote (cons “one” 1)))
```

returns:

```
(“one”. 1)
```

We see that there are three eval functions referenced in this assignment. Here, you implement the eval function entered by the user of your interpreter. This eval calls the predefined eval method

on a list s-expression object. This eval method, in turn, calls the eval method of the evaluator object associated with the first item in the list s-expression. The evaluator eval may be predefined or implemented by you. Thus, a user or s-expression eval call can result in a call to one of many evals defined by registered evaluator objects.

Load

The load operation assumes its argument is a string denoting a file, reads the file into memory, and calls

```
public void newInput(String line)
```

in the model of the lisp-interpreter model, which is an instance of main.lisp.interpreter.ObservableLispInterpreter, The singleton of the model can be retrieved by invoking the get() method in main.lisp.interpreter.InterpreterModelSingleton. There are many libraries for reading file lines - the java.nio.file.Files.readAllLines() alternative is perhaps the simplest to use. The function returns T if it detects no error in the argument or evaluation, and nil otherwise. The basic operations will throw an IllegalStateException if any errors are detected. This exception will be passed down the call stack to the model's newInput method, thus allowing detection of errors in the loaded file.

While the interpreter can accept expressions split between lines and multiple expressions per line, this does not apply with the load operation. Any usage of load MUST be alone on its line or the interpreter will handle it incorrectly. While this is valid for all other operations, with load it will cause out-of-order execution issues.

OK: (load "test.lisp")
OK: (+ 1 2) (eval (quote (+ 2 3)) (+ 3 4)
BAD: (+ 1 2) (load "file.lisp") (+ 2 3)

This specification does not account for errors while running the loaded file. As we allow you to assume that all provided lisp will be valid, you can safely ignore this case. However, if you want to handle it, the built-in operations throw IllegalStateExceptions when errors occur and you can catch them when coming from the InterpreterModel. In the event of an error in the loaded code, load returns nil.

The cond operation takes an ordered list of (<Boolean Expression> <Expression>) sublists, which we will refer to as cases, as an argument. Thus, the list is of the form:

(
(<Boolean Expression^{1}> <Expression^{1}>)
…
(<Boolean Expression^{N}> <Expression^{N}>)

The return value of cond is (the evaluation of) <Expression^{i}> if <Boolean Expression>^{i} == true and $\forall j < i$, <Boolean Expression>^{j} == false. If all expressions are false, it returns nil.

Relational

Operator	Parameters	Result	Example
>	2 (a b), both integer or decimal	a > b ? T : nil	(> 1 2) -> nil (> 3 2) -> T (> (+ 1 2) (+ 1 1)) -> T
<	2 (a b), both integer or decimal	a < b ? T : nil	(< 1 2) -> T
>=	2 (a b), both integer or decimal	a >= b ? T : nil	(>= 1 2) -> nil
<=	2 (a b), both integer or decimal	a <= b ? T : nil	(<= 1 2) -> T

Logical

Operator	Parameters	Result	Example
and	2 (a b)	(all not nil) ? (last non nil) : nil	(and 1 2) -> 2 (and nil 2) -> nil
or	2 (a b)	(any not nil) ? (first non nil) : nil	(or 1 2) -> 1 (or nil 2) -> 2 (or nil nil) -> nil
not	1 (a)	(a == nil) ? T : nil	(not 1) -> nil (not nil) -> T

Expressions

Operator	Parameters	Result	Example
quote	1 (a)	a	(quote 1) -> 1 (quote (+ 1 2)) -> (+ 1 2)
eval	1 (a)	Result of evaluating the value of a	(eval 1) -> 1 (eval (+ 1 2)) -> 3 (eval (quote (+ 1 2))) -> 3
list	Any (a b … z)	List containing a, b, … z	(list) -> nil (list 1) -> (1) (list 1 2 3 (+ 2 2)) -> (1 2 3 4)

cond	>= 1 tuples (a b … z) Where a...z are of the form (a0 a1)	For the first tuple a such that a0 is true, return a1. If none, return nil.	(cond (T 1)) -> 1 (error recovery, not required) (cond (T)) -> T (error recovery, not required) (cond (nil 1) (T 2)) -> 2 (cond (nil 1) ((+ 1 2) 2)) -> 2 (cond ((car nil) 1)) -> nil (error recovery not required)

Task 5: Implementing the OperationRegisterer Interface

As mentioned above, you will need to write your own class implementing main.lisp.evaluator.OperationRegisterer to register your new evaluators with the interpreter and call its registration method in your main method. Change your main method to call the registerOperations() method of it before calling the skeleton main and test your main with expressions containing your new operations. There are currently no tests for directly checking the correctness of your operations, which is indirectly checked through the test lisp file (see below).

Task 6: Implementing the ClassRegistry Interface

As hinted earlier, in addition to registering your s-expression class with the expression factory and your evaluator classes with the operations manager, for grading purposes, you must do the following. Implement a class that implements the predefined interface main.ClassRegistry, and provide appropriate getter methods defined by the interface. You must provide a single class implementing the interface. Our grader will find this class, instantiate it, and call the getter methods to find your classes. It is particularly important to specify your main class so the grader can invoke it to test the functionality.

Task 7: Test Lisp Code

Create a single lisp file, ending with the suffix .lisp (e.g. test.lisp), with expressions that tests each aspect of the functionality you tested. For example, your file must contain an input expression that creates a list to test your toString() method. Like code, tests should not be borrowed from others. They should also not be borrowed from the test cases in localchecks.

Our tests will input (load <filename>) to check your tests for adequacy. The test file should test all operations you implemented except load. There should be no errors in the expressions in the file. That is, when we execute (load <filename>), the result should be T.

One adequacy constraint is that each of your implemented functions is called at least twice and all invocations of the function do not return the same result. Thus, if you submit:

(cons 2 3)
(cons 5 nil)
(list 1 2)
(list 1 2)
(eval (quote (+ 1 2)))

The cons tests pass, but the list, quote and eval tests (partially) fail.

Another adequacy test has to do with the cond function. Make sure you have at least three tests with this operation, each test has at least three cases, and the three tests return the expression in the first, last, and a middle case, respectively.

The file should be in the project folder, not in bin or src. There should be a single file with the suffix .lisp in the project folder.

Once you have created the lisp file, run localchecks again. These will check the adequacy of your tests and indirectly the operations.
All of these are in the package main.lisp.parser.terms. (Currently the source code for them is not in the skeleton github project).

Submission Instructions

As always submit to Gradscope and Sakai. Enclose screenshots of localchecks and also loading the test file. Be sure to include in the zip the test lisp file, which as mentioned above, should be in your project folder.

Possible Gotchas

Add text here on some aspect of the assignment that caused you difficulty and how you resolved it.

-Copying the main code from the Lisp Interpreter into your main - this is code duplication and will fail if we update the LispIntepreter main. Your main can and should call the lisp interpreter.

- Make sure in your class registry to include the main class you made which calls your operations register and registers your basic expression. When running local checks you do not need to have any calls to register anything.

- Try and avoid as much as possible editing the path argument in your load evaluator as there can be unexpected consequences with pathing. If in gradescope you notice that your load check is failing this may be due to you invalidating the path. Additionally if you try and validate the file argument it's best to avoid trying to implement your own check instead rely on java.io.File and use the existing method.