Block 4

Block 4

4-OV Overview

4-OV.1 Contents of This Block

CP This block will discuss liveness, performance and fairness of multithreaded applications.

FP In this block, we will deal with code generation in a functional way, and we will define a parser generator in which a grammar is a function.

LP In this block, the Logic Programming (LP) strand will start. You will learn basic logic programming with Prolog (facts, rules, queries, backtracking, lists, data structures, recursion). This block LP contains three exercises: The Royal Family (§4-LP.1, May 17), The Monkey and Banana puzzle (§4-LP.2, May 17), and the Treesort Algorithm (§4-LP.3, May 19).

CC This block will discuss several forms of intermediate representation: Control Flow Graphs, Dependence Graphs, ILOC with Static Single Assignment, and symbol tables.

4-OV.2 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 4 hours self-study to prepare the CP exercises;
- 2 hours self-study to continue with the LP exercises; you are supposed to finish all three exercises of this week.

Moreover, note that the CC homework exercises are due on the first day after this block.

4-OV.3 Materials for this Block

- **CP**, from JCIP: Chapters 10 and 11.
- **CC**, from EC: Chapter 5.

4-CP Concurrent Programming

Thread dumps. Section 10.1 and 10.2 of JCIP discuss several examples that can cause deadlocks under certain conditions. As explained in Section 10.2.2 of JCIP, JVM's thread dump mechanism can be used to investigate the source of deadlocks.

Exercise 3-CP.1 *Learning goal:* identifying causes of deadlocks.

Listing 10.1 contains a straightforward lock-ordering deadlock. Develop a multi threaded Java program which makes use of the class `LeftRightDeadlock` of Listing 10.1 and observe that it indeed may lead to a deadlock. Use Java's thread dump mechanism to analyse the source for the deadlock. Explain the thread dump information, and how you can derive the source of the deadlock from it. □

Performance optimization of a multi-threaded queue. Buffers are an important concept in concurrent applications, used to distribute work evenly. As they are so ubiquitously spread, they deserve some attention in order to make them as fast as possible and to minimize the chance that the buffer acts as a bottleneck under a very high amount of concurrent accesses.

Exercise 3-CP.2 *Learning goal:* understanding how implementation choice impact data structure performance.

Recall the `Queue` interface from Block 2.

```
public interface Queue<T> {
    /** Push an element at the head of the queue. */
    public void push(T x);

    /** Obtain and remove the tail of the queue. */
    public T pull() throws QueueEmptyException;

    /** Returns the number of elements in the queue. */
    public int getLength();
}
```

1. Start out with your *linked list* implementation of the `Queue` of block 2. You probably used Java's *monitor* pattern to make the implementation thread safe, i.e., declared the methods of the class as `synchronized` methods.
2. Update your test class of Block 2 to add timing measurements to your test framework. Also ensure yourself that your *linked list* implementation is still thread safe.
3. Implement the `Queue` interface with an `java.util.concurrent.ConcurrentLinkedQueue`. Minimize the use of (intrinsic or explicit) locks as much as possible, and ensure that your implementation is thread safe.
4. Implement the `Queue` interface with a `MultiQueue` class. In this `MultiQueue` class each producer has its own `linked list` queue. Consumers will try to obtain an element from any of the producer queues. Again, minimize the use of (intrinsic or explicit) locks as much as possible, and ensure that your implementation is thread safe.
5. Compare the three implementations on performance. How does your implementations of `Queue` scale with the amount of consumers and producers, and the number of insertions per producer. What causes the behavior that you observe? What are the bottlenecks of your implementations? Is there room for further improvement?

□

Copy-on-write. There are many scenarios in which the ratio between reads and writes of a data structure skews heavily in favour of reading. In these cases it might be advantageous to use a *copy-on-write* data structure. The principle of operation is that every *read* of information can be done directly. However, when you want to *write* to the structure the process is a bit more complex:

- Acquire the write lock.

- Make a *deep copy* of the main data structure into thread-local memory. This means that you also need to copy all other objects referenced by the data structure.
- Perform the write operation on the local object. For some structures (e.g., arrays) this might only modify a single object. For other, more complicated objects, more work might be involved. For example, a single write in a balanced tree might trigger a rebalance, thereby possibly affecting the entire tree.
- Replace the main structure by the updated version (i.e., with the write operation applied).
- Release the write lock.

Exercise 3-CP.3 *Learning goal:* understanding how choice of underlying data structures can impact performance of an application.

1. Implement a *copy-on-write* list. Implement at least the methods `add`, `set` and `get`.
2. Compare the performance of your implementation with a `Collections.synchronizedList` on the basis of an `ArrayList`. Compare the performance of your implementation with the class `CopyOnWriteList` from the Java concurrency library. Can you explain your results?
3. Why is there no `CopyOnWriteMap` present in the Java library?

□

Fairness in locks. Sometimes, an important requirement of a concurrent system controlled by locks can be that it is *fair*. The most common meaning of fairness is that every thread gets the chance to do its work eventually. This is a very weak notion of fairness and it should hold for every correct program (otherwise some threads would never be able to make progress). Another notion of fairness is that the thread that requests access first, will also be allowed access first (like a FIFO queue). It is a form of fairness but you could wonder whether it is really fair. What if thread `1` only spends a very short amount of time in the critical section per access but thread `2` spends a lot of time in the critical section per access. In the end thread `2` will have spent a lot more time there than thread `1`, mostly blocking thread `1` from access. Is this fair?

Other possible definitions of fairness are that every thread gets the same amount of time to execute code in the critical section, or that attempted entries with a special high priority flag get their access earlier than lower priority computations.

Exercise 3-CP.4 *Learning goal:* understanding different notions of fairness.

In this exercise, we will implement a fair lock, for your own notion of fairness.

- Pick one of the previously mentioned (or come up with your own) definitions of a relatively strong form of fairness, i.e., not just that all threads eventually get to run their code.
- Why is the fairness definition you picked fair? In what scenarios would it not be fair? Can you come up with an example scenario in which this fairness definition would be useful?
- Modify one of the synchronizers you created in Exercise 2-CP.4 in order to make it behave according to your definition of fairness.

□

4-FP Functional Programming

See BLACKBOARD.

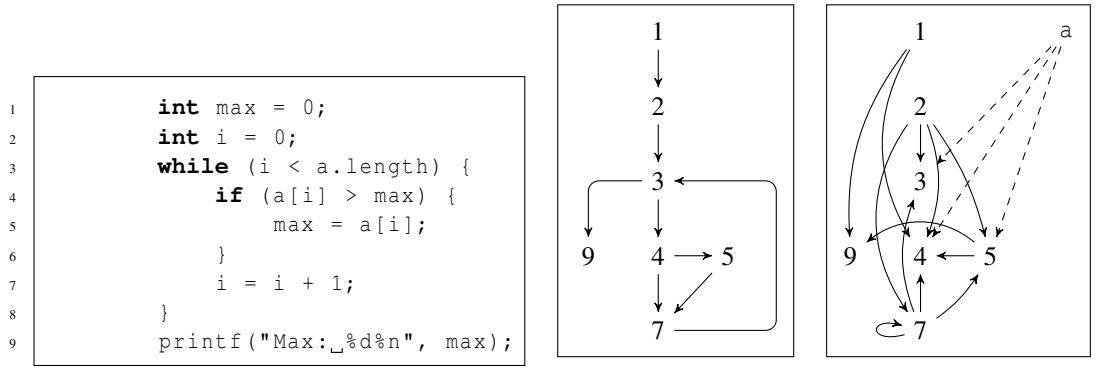

Figure 4.1: Control flow (b) and dependency graph (c) of a JAVA fragment (a)

4-CC Compiler Construction

4-CC.1 Control flow and dependency graphs

Exercise 4-CC.1 Compile for yourself a list of the following terms, with short definitions for each of them. Don't just copy from the book: make sure you understand your own definitions.

AST (esp. difference with parse tree), DAG, basic block, CFG (two meanings!), dependence graph, call graph, SSA, symbol table

□

Consider the program fragment in JAVA in Figure 4.1(a), which finds the maximum element in an array `a` of non-negative integers. The control flow graph and dependency graph of this fragment are depicted as (b) and (c). The numbers in the figure correspond to the line numbers of the fragment. From the control flow graph, it can be seen that only 1 and 2 may be combined into a basic block, as all other nodes either have more than one incoming edge or more than one outgoing edge. From the data flow graph, it can be seen that within that block, statements 1 and 2 may be reordered, as there is no dependency between them.

Exercise 4-CC.2 Consider the following code fragment:

```
1  int x = in();
2  int i = 0;
3  boolean found = false;
4  while (!found && i < a.length) {
5      found = (a[i] == x);
6      if (!found) {
7          i = i + 1;
8      }
9  }
10 printf("Index: %d%n", i);
```

1. Draw the control flow graph and the data dependency graph.
2. According to the control flow graph, what are the basic blocks of this fragment?
3. According to the data dependency graph, which statements can be reordered? □

Exercise 4-CC.3 Consider the following code fragment:

```
1  int up = in();
2  int sum = 0;
3  for (int i = 0;
4           i < up;
5           i = i + 1) {
6      if ((i & 1) == 0) {
```

```
7                   continue;
8               }
9               sum = sum + i;
10          }
11          printf("Sum: %d%n", sum);
```

If you have never seen the **continue**-statement, look it up!

1. Draw the control flow graph and the data dependency graph.
2. According to the control flow graph, what are the basic blocks of this fragment?
3. According to the data dependency graph, can any statements be reordered? □

In the next exercise you will write a tree visitor to build control flow graphs. Some code has been provided in the package `pp.block4.cc.cfg`:

- `Graph.java`: a simple control flow graph
- `Node.java`: node of a control flow graph, with fields for ID and number.
- `Fragment.g4`: Grammar for a fragment of JAVA that will compile the code snippets above.
- `BottomUpCFGBuilder.java`: Unfinished template for Exercise 4-CC.4
- `TopDownCFGBuilder.java`: Unfinished template for Exercise 4-CC.4

Control flow graphs can be constructed through top-down, inheritance-based rules or bottom-up, synthesis-based rules. In either case, the idea is that every parse subtree adds control flow nodes and edges to a global graph (given as an instance variable), but there is a `ParseTreeProperty` associating the correct entry and exit nodes.

Bottom-up: This is the most straightforward way. Every `exit`-method of the listener can build a control flow graph, based on the CFGs of its sub-statements. The construction is very much like that of an NFA from regular expressions. For simplicity, make sure that you stick to the convention given in the book that every CFG has a single entry point and a single exit point; then you only have to store the entry and exit points as attributes in your listener. The choice of having a single exit point will sometimes necessitate the creation of "fake" control flow nodes that do not correspond to any statement but just serve as exit nodes of some sub-CFG. Thus, the graphs constructed in this way will not be identical to the hand-drawn ones of the previous exercises.

Top-down: This is more tricky. Every statement tells all its children the entry and exit nodes they should use. For this purpose, the `enter`-methods of the listener should prepare those nodes and set them as attributes for their children. The `exit`-methods are unused.

Exercise 4-CC.4 Now program both solutions. *You may omit the **break** and **continue** statements; they are addressed in a challenge exercise below.*

1. Program the bottom-up CFG builder (template provided in `BottomUpCFGBuilder`)
2. Program the top-down CFG builder (template provided in `TopDownCFGBuilder`)
3. For both builders, show that they return correct flow graphs for the code snippets of Figure 4.1 above and Exercise 4-CC.2 (note that you do not have to declare array `a`). It is expected that the graphs are not *identical* to your answers to those exercises. What are the differences? □

Challenge JAVA knows the concept of *abrupt termination*: this is what happens if you **break** out of a loop or **continue** within that loop, and also if you **return** anywhere except at the end of a method body or when an exception gets **throw**n. All such phenomena disrupt the normal flow of control (and most are actually frowned upon by code purists for that reason); to get the CFG correct for such cases takes special care.

Exercise 4-CC.5 *This exercise is meant as a challenge and does not have to be signed off.*

Extend your bottom-up or top-town CFG builder (or both) from Exercise 4-CC.4 with methods for **break** and **continue** statements (both of which are already in the provided grammar `Fragment.g4`). For this purpose, you should realise that in fact every type of abrupt termination essentially requires its own exit node; those exit nodes can be synthesized or inherited in essentially the same way as the "regular" exits. Test out your program on the the JAVA snippets of Exercise 4-CC.3 and on the following (which computes the same value as the snippet of Exercise 4-CC.2):

```
int x = in();
int i;
for (i = 0; i < a.length; i++) {
    if (a[i] == x) {
        break;
    }
}
printf("Index: %d%n", i);
```

□

4-CC.2 ILOC

Appendix A of EC contains a specification of the linear intermediate representation format ILOC used in the book. Examples of the use of ILOC are scattered throughout the book. For instance, Fig. 5.8(b) shows how an array element is assigned in ILOC, and Fig. 5.14 shows more complicated ILOC code fragment. The JAVA snippet in Figure 4.1 is translated to the following ILOC code[1], in which `@a` denotes the offset of the beginning of array `a` from the address pointed to by `r_arp`, `@alength` denotes the length of the array `a`, and all other variables are kept in registers.

```
1  start: loadI 0 => r_max           // Line 1: max = 0;
2         loadI 0 => r_i             // Line 2: int i = 0;
3         loadI @alength => r_len
4  while: cmp_LT r_i,r_len => r_cmp  // Line 3: while (i < a.length)
5         cbr r_cmp -> body, end
6  body:  i2i r_i => r_a             // compute address of a[i]
7         multI r_a,4 => r_a         // multiply by size of int
8         addI r_a,@a => r_a         // add a's base offset
9         loadAO r_arp,r_a => r_ai   // r_ai <- a[i]
10        cmp_GT r_ai,r_max => r_cmp // Line 4: if (a[i] > max)
11        cbr r_cmp -> then,endif
12 then:  i2i r_ai => r_max          // Line 5: max = a[i];
13 endif: addI r_i,1 => r_i          // Line 7: i = i + 1;
14        jumpI -> while
15 end:   out "Max: ", r_max         // Line 9: out; not "official ILOC"
```

In the lab files you will find an assembler, a disassembler and a virtual machine for ILOC:

- `pp.iloc.Assembler` is an assembler. It has a singleton instance that; the method `parse` reads in ILOC programs in textual form (given as a string or in a file), such as the one shown above, and returns an object of type `pp.iloc.model.Program`.
- The method `prettyprint` of the class `Program` returns a string representation of the program, in the original syntax of ILOC.
- `pp.iloc.Simulator` simulates a `Program` by running it on a virtual machine (`pp.iloc.eval.Machine`) consisting of a simulated block of memory, a register map, and values for the symbolic constants `@a` etc.
- `pp.iloc.test` contains examples of the use of `Assembler` and `Simulator`.

Exercise 4-CC.6 Try out the assembler, disassembler and simulator on `max.iloc` by writing a JUNIT test with the following test methods:

1. A test method that runs the assembler on `max.iloc` and calls `prettyprint` on the resulting `Program` object. Test that, if you parse the prettyprinted program again, the resulting `Program` equals the one you got by parsing the original `max.iloc`.
2. A test method that run the resulting `Program` in the `Simulator`. The method `Simulator#run` starts off the simulation; however, before calling it you should initialize the VM (which you can access through `Simulator.getVM`) by initializing the array `a`, i.e., giving values to the constants `@a` and `@alength` using `Machine#init` and `Machine#setNum`, respectively. Test that you get the expected output by programmatically inspecting the value of the register `r_max` after the simulation has finished, using `Machine#getReg`.

[1] Available in the lab files of Block 4 as `pp/block4/cc/iloc/max.iloc`

3. Explain the multiplication by 4 in Line 7 of the program. □

In the ILOC program above, the *basic blocks* correspond to the fragments starting with a label.

Exercise 4-CC.7 Draw the CFG of the ILOC program above, using the basic blocks as nodes, and compare it to the one in Figure 4.1. □

Exercise 4-CC.8 Answer Review Question 1 of Section 5.4 of EC. Ignore the fact that `fib` is a function: just implement the function body.

1. Write an ILOC program implementing `fib` based on a register-to-register model.
2. Write an ILOC program implementing `fib` based on a memory-to-memory model.
3. Write a test that shows that both programs return the same value for a range of input values of your choice. (For this purpose, you have to appropriately initialize the VM and inspect the result afterwards.)
4. What is the largest value for `n` for which the programs do not produce an integer overflow? □

Exercise 4-CC.9 Write ILOC code for the JAVA fragment in Exercise 4-CC.5. (It does not matter if you did not do that exercise.) To mimic the `in()`-call in the JAVA program, use the pseudo-ILOC-instruction **`in`**; see `pp.iloc.OpCode.in`. (Like the **`out`**-instruction demonstrated in the ILOC code fragment, this instruction is not part of the "official" ILOC: it is only provided for debugging purposes.)

Make sure your solution passes the provided `pp.block4.cc.test.FindTest` (which assumes that your code can be found in `pp/block4/cc/iloc/find.iloc`; change that to the correct filename). □

Finally, in the following exercise you will do some actual code generation (of ILOC code). This will be the first real compiler you build! Use the attribute rules in Fig. 4.15 (Page 207) of EC as inspiration.

Exercise 4-CC.10 Extend `pp.block4.iloc.CalcCompiler`, a tree listener for parse trees produced by the grammar `pp/block4/cc/iloc/Calc.g4` (which is actually identical to `pp/block3/cc/antlr/Calc.g4` of the previous Block). `CalcCompiler#compile` should generate ILOC code (in the form of a `Program` object) for an arbitrary `Calc`-expression, using a register-to-register memory model, ending with an `out` operation to print the calculated value. For instance, for the expression $1 + -3 * 4$, the generated code should be something like

```
loadI   1                => r_0
loadI   3                => r_1
loadI   4                => r_2
mult    r_1,r_2          => r_3
rsubI   0,r_3            => r_4
add     r_0,r_4          => r_5
out     "Outcome: ",r_5
```

Make sure your solution passes the `pp.block4.cc.test.CalcCompilerTest`. □

4-LP Logic Programming

The idea is to make the first two sections (Royal Family, Monkey Banana) on May 17, and have the last exercise from each section signed off. The third section (Tree Sort) is meant for May 19, including the self study.

4-LP.1 Royal Family: Facts, Rules and Queries

If you have not done so: Install SWI Prolog, following the instructions under Tools/Installing Prolog, and find the program "royal_family_facts.pl" on Blackboard (lab files Block 4). You can load this program from the Prolog prompt by: `?- consult(royal_family_facts).`

Exercise 4-LP.1 Start a new program "royal_family_defs.pl" and enter the rules from the lecture for child/2, grandparent/2. Test if this works. □

Exercise 4-LP.2 Provide precise definitions of several family relations, like: brother/2, aunt/2, cousin/2, nephew/2. □

Exercise 4-LP.3 Test your definitions with a number of test instances. Explain the answers, as well as the number of repetitions that you see. □

Exercise 4-LP.4 Provide your own definition of ancestor/2. Show that this definition might loop, by modifying the facts database "royal_family_facts.pl".

4-LP.2 Monkey gets Banana: Functions, Backtracking, Lists

The "Monkey gets Banana" problem is modeled as `state(Monkey,Pos,Box,Has)`, where `Monkey` and `Box` indicate the location of the monkey and the box in the room, respectively (at the door, at the window, or in the middle); `Pos` indicates the position of the monkey (on the box or on the floor) and `Has` indicates if the Monkey has already got the banana.

Exercise 4-LP.5 Define a predicate `goal/1` that is true for all states where the Monkey has the Banana. □

Exercise 4-LP.6 Define a predicate `init/1` that is true for the initial state, where the Monkey is on the floor at the door, without having the banana, and the box is at the window. □

Exercise 4-LP.7 Define the move-predicate `move/3`, where `move(State1,Move,State2)` indicates that action `Move` brings the monkey from `State1` to `State2`. The possible moves are:

1. *climb*: If the monkey and the box are in the same location, and the monkey is on the floor, it can climb on top of the box.
2. *grasp*: If the monkey and the box are in the middle and the monkey is on the box without a banana, it can grasp for the banana and get it.
3. *walk(L1,L2)* If the monkey is in location L1 on the floor, it can walk to position L2
4. *push(L1,L2)* If the monkey and the box are in location L1 and the monkey is on the floor, it can move with the box to location L2.

□

Exercise 4-LP.8 Write a general problem solver `solve/1`, which can use `move/3` and `goal/1`.
`:- init(S), solve(S)` shall check if the goal can be reached from the initial state by a number of moves. □

Exercise 4-LP.9 Experiment with the order of the `move/3` clauses in the program. How does the order affect termination of the program? □

Exercise 4-LP.10 Extend the solver `solve/2` to return a solution: `init(S), solve(S,L)` should return a list of moves L, e.g.:
`yes. S=... and L=[walk(atdoor,atwindow),climb,...].`

Exercise 4-LP.11 *Optional:* Extend the solver to print intermediate states, so you can get some insight in Prolog's search and backtracking. You can use the built-in predicate `print/1` to print a term, and `nl/0` to print a new-line. □

4-LP.3 Tree Sort: More on lists, Data structures, Recursion

A binary tree is a datastructure in which each node has a value and two children. In this assignment, we define a binary tree as either `nil` or `t(L,N,R)`, where `nil` denotes the empty tree and `L` and `R` are trees again, called the left subtree and the right subtree respectively; and `N` is a term. You may suppose that `N` always is a number, without checking this explicitly.

A binary tree is sorted if for each node (subtree) `t(L,N,R)` in the tree:

- The left subtree `L`, if not nil, has a maximum smaller than or equal to the value `N`; and
- The right subtree `R`, if not nil, has a minimum greater than or equal to its own value `N`.

The idea is to make exercises 4-LP.12 – 4-LP.20, and signoff only exercise 4-LP.20 with a practicum assistant. The last two exercises 4-LP.21 – 4-LP.23 are optional, for those who like an extra challenge.

Exercise 4-LP.12 Write a predicate `istree(T)` that evaluates to true iff T is a binary tree. Example:

```
?- istree(
   t( t(nil,2,nil),
      4,
      t(  nil,
          5,
          t(nil,18,nil)
 )  )  ).
Evaluates to True.

?- istree(t(t(2,2),4,nil)).
Evaluates to False.
```

□

Exercise 4-LP.13 Write the predicate `max(T,N)` and `min(T,N)` that find respectively the maximum value and minimum value of a sorted binary tree. Example:

```
?- min(t(t(t(nil, 1, nil), 2, nil), 3, nil), R).
R=1
?- max(t(t(t(nil, 1, nil), 2, t(nil, 8, nil)),
                          18, t(t(nil, 21, nil), 81, t(nil, 218, nil))),R).
R = 218
```

□

Exercise 4-LP.14 Write a predicate `issorted(T)` that evaluates if the argument of the predicate is a sorted binary tree (see the explanation above for a definition). Example:

```
?- issorted(t(t(nil,4,nil),5,t(nil,8,nil))).
true
?- issorted(t(t(nil,2,nil),1,nil)).
false.
```

□

Exercise 4-LP.15 Write a predicate `find(T, N, S)` that finds a node with a specific value N in a sorted binary tree T and returns the corresponding subtree in S. It fails if such a node does not exist. Example:

```
?- find(t(t(nil,1,t(nil,3,nil)),5,t(t(nil,6,nil),8,nil)),1,S).
S = t(nil, 1, t(nil, 3, nil)).
```

□

Exercise 4-LP.16 Write a predicate `insert(T, N, S)`, which inserts a value N in the sorted binary tree T, giving sorted binary tree S. Example:

```
?- insert(t(t(nil,1,t(nil,3,nil)),5,t(t(nil,6,nil),8,nil)),7,S).
S = t(t(nil, 1, t(nil, 3, nil)), 5, t(t(nil, 6, t(nil, 7, nil)), 8, nil)).
```

□

Exercise 4-LP.17 Write a predicate `deleteAll(T, N, S)` that deletes one value N from a sorted binary tree T, giving sorted binary tree S. Hint: Use the result of assignment 2. It might be easier to use an auxiliary function that deletes only one element. Example:

```
?- deleteAll(t(t(nil, 1, t(nil, 3, t(nil,3,nil))), 5, t(t(nil, 6, nil), 8, nil)),3,S).
S = t(t(nil, 1, nil), 5, t(t(nil, 6, nil), 8, nil)).
```

□

Exercise 4-LP.18 Write a predicate `listtree(L,T)` where input L is a list of integers, and output T is a sorted binary tree with the same values. Hint: it might be helpful to use `insert`. Example:

```
?- listtree([8,3,10,2,11,7,1],T).
T = t(t(t(t(nil, 1, nil), 2, nil), 3, t(nil, 7, nil)), 8, t(nil, 10, t(nil, 11, nil))).
```

□

Exercise 4-LP.19 Write a predicate `treelist(T,L)` where input T is a sorted binary tree and output L is a sorted list of unique integers with the values from T. Hint: it might be helpful to use `deleteAll` Example:

```
treelist(t(t(nil,1,nil),2,t(nil,8,nil)),L).
L = [1, 2, 8]
?- treelist(t(t(nil,1,t(nil,3,nil)),3,t(t(nil,6,nil),8,nil)),L).
L = [1, 3, 6, 8].
```

□

Exercise 4-LP.20 Write a predicate `treesort(L1,L2)` which succeeds if output L2 is the sorted permutation of input L1. The sorting algorithm should use a binary sorted tree. Test the algorithm with a number of test instances. What happens if L1 contains duplicate entries? Explain why this happens.

Optional Extension

A binary tree is balanced if, for each node, the number of nodes of the left subtree and the number of nodes of the right subtree differ at most 1. We only use this concept in the optional excercises.

Exercise 4-LP.21 *(optional).* Write the predicate `listtree_balanced` such that the resulting tree is balanced. Use this to write `balanced_treesort`. Compare the complexity of the two sorting algorithms by measuring the runtime on a number of larger examples. □

Exercise 4-LP.22 *(optional).* Write a predicate to write a tree graphically to your screen.

Example (but you can decide on another tree-layout yourself):

```
listtree_balanced([8,3,10,2,11,7,1],T), draw(T).
         7
        / \
       /   \
      /     \
     /       \
    2         10
   / \        / \
  /   \      /   \
 1     3    8    11
T = t(t(t(nil,1,nil),2,t(nil,3,nil)),7,t(t(nil,8,nil),10,t(nil,11,nil))).
```

□

Exercise 4-LP.23 *(optional).* Try to run `insert` in reverse. Is it possible to get `delete` for free? Similar, can the computation of `treelist` be obtained as the inverse of `listtree`? Maybe after modifications to the program? □