Block 5

# Block 5

## 5-OV Overview

### 5-OV.1 Contents of This Block

**CP** This block discusses lock-free algorithms and data structures.

**CC** This block will discuss issues related to procedure calls and memory layout: how should local variables, global variables and parameters be stored and accessed so as to achieve the well-known call mechanism?

### 5-OV.2 Expected Self-Study and Project Work

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

- 6 hours self-study to prepare the CP exercises;
- 2 hours self-study to read through the CC material and complete the lab exercises.

### 5-OV.3 Materials for this Block

- **CP**, from JCIP: Sections 14.5 and 14.6, and Chapter 15.
- **CC**, from EC: Chapter 6.

## 5-CP Concurrent Programming

**Lock-free programming.** Besides solving many concurrency-related problems (e.g., data races), locks can also introduce a wide range of problems (e.g., deadlocks and performance degradations). However, there are ways to make containers thread safe without the use of locks, so-called *lock-free* programming (aka lockless program). A program is called lock-free if it satisfies the following property: as long as the program is able to keep calling lock-free operations, the number of completed lock-free operations keeps increasing, no matter what. It is algorithmically impossible for the system to lock up during those operations. As always, a new method of solving existing problems is prone to introduce new problems.

**WARNING.** Lock-free programming is incredibly tricky to get right. It is strongly recommended to go online, look up sample implementations, possible problems (look for the ABA problem) and their possible

solutions. The correctness of lock-free containers is even more difficult to check and guarantee than when using locks – the best way to learn is to look up the mistakes and solutions of others who have spent a lot of time thinking about the consequences of their design choices and tried to make sure their implementations were thread-safe. Some example information to get you started:

- A quick overview of lock-free programming problems
- Some more elaboration on the ABA problem.

**Exercise 5-CP.1** In this exercise, we will look at lock-free stack implementations.

1. One of the most simple data structures to make lock-free is a stack. During the lecture we discussed Treiber's stack. Implement a lock-free stack yourself, featuring both a lock-free `push` and `pop` operation. Make sure that your stack implements the following interface:

```
public interface LockFreeStackInterface <T> {
    public void push(T x);
    public T    pop();
    public int  getLength();
}
```

2. As always: devise a test, and apply it to make sure that your lock-free implemenation works as expected.
3. Can you adjust your implementation to change it into a bounded stack? If yes, explain how you do this. If not, why is this not possible?
4. Make note of the issues that can arise from lock-free programming and try to identify ways to detect and prevent these problems. Note: some of the problems you encounter will be solved by the garbage collector as they only revolve around occasionally losing references, but often the problems are more complex; you also have to take them into account when programming in Java.

□

**Barrier implementation.** Last week you have seen an application of a barrier – as part of a MAPREDUCE framework. Often, especially when writing *parallel* programs (in contrast to writing *concurrent* programs) the need for barriers will arise because of the need to synchronize the progress of different threads. As with locks, there are many ways to implement a barrier, in both locked and lock-free ways.

Consider the interface `Barrier`:

```
public interface Barrier {
    public int await() throws InterruptedException;
}
```

**Exercise 5-CP.2** In this exercise, we will look at various ways to implement barriers.

1. Implement a barrier with locks and a `wait/notify` or `await/signal` system. The use of `wait/notify` has been discussed during the lecture of block 4. Furthermore, some additional resources on writing code with `wait/notify` can be found here:
   - Why `wait`, `notify` and `notifyAll` must be called from synchronized block or method in Java.
2. Implement a barrier using lock-free programming. Similarly to Exercise 5-CP.1, think very carefully about your design as lock-free programming can introduce very subtle bugs. However, by using some of the classes in `java.util.concurrent` you can simplify your design a lot – decreasing the odds of concurrency bugs.

□

**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?

Another 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 5-CP.3** 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 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 4-CP.1 in order to make it behave according to your definition of fairness.

□

# 5-FP Functional Programming

Description of Functional programming, block 5

# 5-CC Compiler Construction

## 5-CC.1 ANTLR tree visitors

So far, we have used ANTLR to define *tree listeners*. You have seen that a tree listener combines a pre-order and a post-order traversal of the parse tree. In some cases, however, this is not powerful enough: it can happen that one needs an in-order traversal (where the parent node is visited after the first child or in between every pair of children) or even another, more dedicated traversal strategy. For this purpose, ANTLR also supports *tree visitors*. Tree visitors require a bit more work from the programmer but offer more control: essentially, they allow you to completely define your own traversal strategy.

The lab files for this block include a grammar `Building.g4` to illustrate the principle of tree visitors. To generate a tree visitor for this grammar, invoke ANTLR on `Building.g4` with the command-line arguments `-no-listener -visitor`. (See BLACKBOARD for directions on how to do this.) This will generate classes `BuildingVisitor` and `BuildingBaseVisitor` rather than the `BuildingListener` and `BuildingBaseListener` we have been working with so far.

In contrast to `BuildingListener`, the tree visitor interface `BuildingVisitor` provides not two but a *single* abstract method for every nonterminal: the dedicated method for a nonterminal `x` is called `visitX`. As parameter, each such method gets the same sort of context object as the `enter`- and `exit`-methods of a listener. Moreover, `BuildingVisitor` has a generic type; this is the return type of every `visitX`-method. In implementing a visitor, extend `BuildingBaseVisitor` and override its `visitX`-methods, taking care of the following:

- Each `visitX` should call the global method `visit` on all children that need to be visited, in the order they need to be visited.
- In between the calls to `visit`, each `visitX` can do whatever it likes.

- Each `visitX` method should return a value of the instantiated generic type.

**Exercise 5-CC.1** *This exercise does not need to be signed off; it serves as a preparation for the next ones.* Study the provided `Building.g4` and `Elevator.java`

1. Generate the `Building` visitor classes using ANTLR.
2. Invoke `Elevator.java` on input of your choice. Study the `visitBuilding` and `visitFloor` methods and make sure you understand what they do. Note that `visitRoom` is never called (and hence it is not overridden in this class). □

Next, you'll program your own tree visitor. Consider the situation where an input string consists of an alternating sequence of numbers and words, such as for instance the text

```
3 strands 10 blocks 11 weeks 15 credits
```

and we want to print this on the standard output in the more readable form

```
3 strands, 10 blocks, 11 weeks and 15 credits
```

while simultaneously calculating the sum of the numbers occurring in the sentence (in this case, the sum would be 39). In the lab files, you will find a grammar `NumWord.g4` that can parse the input: the parser rules are

```
sentence: (number word)+ EOF;
number: NUMBER;
word: WORD;
```

(where `WORD` and `NUMBER` are token types with the expected definition).

Neither a preorder traversal nor a postorder traversal provides a natural way to print the required separators ", " and " and " between the groups of `number word`-pairs, so using a tree listener on this grammar it is not straightforward to program the desired behaviour. (It is not *impossible*: the exit method for `word` may inspect its parent and depending on whether that is a `sentence` and on the position of itself within that sentence print the correct separator.)

**Exercise 5-CC.2** Implement the tree visitor `NumWordProcessor.java` (a skeleton of which is provided in the lab files) by overriding the `visitSentence`, `visitNumber` and `visitWord` methods so that the class has the required behaviour, described above (print the converted sentence, and return the sum of the numbers). You can test your `NumWordProcessor` by invoking the (provided) `main` method. □

**Optional exercise: Faking the tree visitor.** Alternatively, the functionality of a visitor can be faked through a listener by adding auxiliary intermediate nonterminals to the grammar, which are traversed by the listener at the right moments. For instance, the following rules, provided in the lab files in the form of the grammar `NumWordGroup`, generate the same language as `NumWord` above:

```
sentence
    : (group* penultimateGroup)? lastGroup EOF;

group
    : number word;

penultimateGroup
    : number word;

lastGroup
    : number word;

number: NUMBER;
word: WORD;
```

Here, the nonterminals `group`, `penultimateGroup` and `lastGroup` (which actually have exactly the same right hand sides) are defined with the sole purpose of allowing an "ordinary" tree listener to distinguish the right places in the sentence for the different separators.

**Exercise 5-CC.3** *(This is an optional exercise and does not have to be signed off.)* Implement the tree listener `NumWordGroupProcessor.java` (a skeleton of which is provided in the lab files) by overriding the appropriate `enter-` or `exit-` methods so that the class has the required behaviour. Again, you can test your `NumWordGroupProcessor` by invoking the (provided) `main` method. □

**Exercise 5-CC.4** *This is a follow-up to the optional Exercise 5-CC.3. It is meant for you to form your own opinion about the different solutions in Exercises 5-CC.2 and 5-CC.3; it does not have to be signed off.*

1. What advantages can you see of the visitor-based solution over the listener-based solution?
2. What advantages can you see of the listener-based solution over the visitor-based solution? □

**Generating ILOC for control structures.** We'll now combine a lot of the ingredients from Block 4 to create an ILOC code generator for a language that starts to look like you could actually do something with it. The provided grammar `SimplePascal.g4` defines a non-trivial fragment of the programming language `Pascal`, with the following features:

- The language supports integer and boolean types. On the ILOC level, booleans may be encoded as integers.
- All variables are defined up front, in a single, global scope. Upon declaration, variables are initialised to `0`, respectively `false` for booleans. Variables should be stored in memory; for this purpose, you have to calculate the offset of every variable with respect to the base of the scope. Remember that an integer uses four bytes.
- Besides assignments and blocks, the language supports `If`- and `While`-statements. This means that you have to generate conditional and immediate jumps (`cbr` and `jumpI`, respectively. For this purpose, you will need the tree visitor functionaltity introduced in Exercises 5-CC.1 and 5-CC.2.

Two sample PASCAL programs are provided together with the lab files:

- `gcd.pascal`, containing Euclid's algorithm for the computation of the greatest common divisor of two numbers;
- `prime.pascal`, containing a naive primehood detection algorithm.

**Exercise 5-CC.5** Write a Simple Pascal program `gauss.pascal` that calculates the sum of all integers up to some upper bound set through an `In`-statement. *This question does not have to be signed off separately; you'll be asked to add this program to the tests of one of the next questions.* □

Because this language is no longer trivial, it becomes necessary to use a *two-pass compiler*. The first pass should achieve the following:

- It takes care of type checking;
- It calculates the offsets of the declared variables;
- It determines the entry points of the flow graphs corresponding to the statement and expression nodes of your parse tree.

The first pass (Exercise 5-CC.6) can be implemented using a tree listener. The results of this pass are collected into a single object that is used in the second pass (Exercise 5-CC.7 below) to generate ILOC code.

**Exercise 5-CC.6** Complete the tree listener `pp.block5.cc.simple.Checker` so that it returns an instance of the class `Result` containing the outcome of the checker phase. Note that all expression-related listener methods have already been given; you have to provide the statement- and declaration-based methods. (Look into the grammar `SimplePascal.g4` to see which methods those are: as usual, everything that (transitively) occurs in the rules for `decl` and `stat` can potentially play a role in the calculation of the attributes.)

Test your solution using the provided `SimpleCheckerTest`. □

The second pass is more conventiently programmed as a tree visitor rather then a tree listener.

**Exercise 5-CC.7** Complete the tree visitor `pp.block5.cc.simple.Generator` so that it generates (correct) ILOC code. Test your solution by running `SimpleCompiler` manually on the provided PASCAL files as well as your own solution to Exercise 5-CC.5, and also by running `SimpleGeneratorTest`.

To use the provided `Generator` optimally, consider the following:

- The generic type of the `TreeVisitor` has been instantiated to `Op`, meaning that every `visit` method returns a value of this type. You can take advantage of this to return the *first instruction* of the code emitted for a given parse tree node, which in turn can help you to find the correct (conditional and immediate) jump targets.
- To generate code for the conditional statements (**`If`** and **`While`**), you can apply the techniques you learned for the (bottom-up or top-down) control flow graph construction in Exercise 4-CC.4 of the previous Block.

□

## 5-CC.2 Dealing with procedures

**Exercise 5-CC.8** Solve Exercise 6.3 from EC. □

**Exercise 5-CC.9** In the PASCAL program of the previous exercise, consider the situation where

- `Main` has called `P1` (Line 33);
- `P1` has called `P4` (line 16);
- `P4` has called `P5` (line 30);
- `P5` has called `P1` (line 25).

Draw a graph showing the activation records at this point (including the one of `Main`), along the lines of Fig. 6.4 and 6.8, for each activation record showing:

- The local data area (containing slots for each of the local variables — you do not have to invent values of the variables);
- The pointer to the caller's ARP;
- The access link (see Fig. 6.8). □

**Exercise 5-CC.10** Solve Exercise 6.6 from EC, taking the following points into account:

- Figure 6.7 from EC gives an example of the kind of answer that is expected here.
- Show a single instance of `Elephant` and a single instance of `Dumbo`; choose your own values for the instance variables.
- The situation in JAVA differs from the one depicted in Fig. 6.7 in that the class of a class object is `Class`, whereas the superclass of every class that does not explicitly declare its own superclass is `Object`. Make sure you depict this situation correctly. □

**Exercise 5-CC.11** Solve Exercise 6.8 from EC for the cases of call-by-value, call-by-reference and call-by-name (in other words, omitting call-by-value-result). For each of the simulations, show the content of the variables `a` and `i` after lines 6, 7, 8 and 9 of the program. You do not have to show the result of the print statements. □

**Exercise 5-CC.12** Solve Exercise 6.10 from EC. Take into account that the ARs are stack-allocated here, and hence you have to actually combine them into a single stack. □

**Exercise 5-CC.13** Solve Exercise 6.11 from EC. □