Block 2

Block 2

2-OV Overview

2-OV.1 Contents of This Block

CP This block will discuss lock-based synchronisation. Both Java's intrinsic locks, as built in to Java by means of the keywords `synchronized`, and explicit locks, provided by the Java concurrency library are discussed.

FP During this block we will introduce some further aspects such as higher order functions, pattern matching, lambda abstraction, list comprehension.

CC In this block we concentrate on the parsing phase of compilation. From EC we will intensively study LL(1) parsing. We'll skip the details of the equally interesting, more powerful but conceptually harder LR(1) parsing algorithm. We then see what ANTLR has to offer in the way of parser rules: quite a bit, as it turns out.

2-OV.2 Mandatory Presence

During the following activities, your presence is mandatory.

2-OV.3 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 study the CC material.

2-OV.4 Materials for this Block

- **CP**, from JCIP: Chapters 2, 4 and 13.
- **CC**, from EC: Sections 3.1–3.3 (intensively), Sections 3.5–3.6 (superficially).

2-CP Concurrent Programming

Before the second exercise session, you should try to solve a collection of self-study exercises. You are asked to develop complete Java test programs, run these programs, and analyse the results of your tests. During the plenary exercise session, you will be asked to present your solutions, and the different solutions will be discussed in the group.

Exercise 2-CP.1 *Learning goal:* identifying concurrency problems.

Consider the class `Hotel` below, which implements some simple hotel functionality, maintaining an array of rooms and a list of people waiting to check in. For simplicity, it does not model that guests can check out; you can assume that a separate thread will take care of this.

```java
import java.util.*;
import java.util.concurrent.locks.*;

class Hotel extends Thread {
    private int nr_rooms = 10;
    private Person[] rooms = new Person[nr_rooms];
    private List<Person> queue = new ArrayList<>();
    private Lock queueLock = new ReentrantLock();

    boolean occupied(int i) {
        return (rooms[i] != null);
    }

    int checkIn(Person p) {
        int i = 0;
        while (occupied(i)) {
            i = (i + 1) % nr_rooms;
        }
        rooms[i] = p;
        return i;
    }

    void enter(Person p) {
        queueLock.lock();
        queue.add(p);
        queueLock.unlock();
    }

    // every desk employee is a thread
    public void run() {
        while (true) {
            if (!queue.isEmpty()) {
                queueLock.lock();
                Person guest = queue.remove(0);
                queueLock.unlock();
                checkIn(guest);
            }
        }
    }
}

class Person {
    // some appropriate query functions
}
```

The implementation has several different (concurrency) errors. Discuss *at least three* different errors, explain why they cause a problem, and what should be done to fix them. □

Exercise 2-CP.2 *Learning goal:* understanding that the preservation of the relation between variables influences locking policies.

Consider the class `Point` and its JML annotations below. It is similar in spirit to the class `Point` that we discussed last week, except that:

- it uses `synchronized` blocks to make sure that the access to the shared variables `x` and `y` is properly protected, and
- it tries to ensure that `x` and `y` never get the same value.

```
public class Point {
    /*@ spec_public */private int x;
    /*@ spec_public */private int y = 1;
    //@ public invariant x != y;

    private Object lockX = new Object();
    private Object lockY = new Object();

    //@ ensures \result >= 0;
    /*@ pure */public int getX() {
        synchronized (lockX) {
            return x;
        }
    }

    //@ ensures \result >= 0;
    /*@ pure */public int getY() {
        synchronized (lockY) {
            return y;
        }
    }

    /*@ requires n >= 0;
     *  ensures getX() == \old(getX()) || getX() == \old(getX()) + n; */
    public void moveX(int n) {
        boolean b;
        synchronized (lockX) {
            synchronized (lockY) {
                b = (x + n != y);
            }
        }
        synchronized (lockX) {
            if (b) {
                x = x + n;
            }
        }
    }

    /*@ requires n >= 0;
     *  ensures getY() == \old(getY()) || getY() == \old(getY() + n); */
    public void moveY(int n) {
        boolean b;
        synchronized (lockX) {
            synchronized (lockY) {
                b = (x != y + n);
            }
        }
        synchronized (lockY) {
            if (b) {
                y = y + n;
            }
            }
        }
    }
```

For all JML specifications of `Point` (i.e., for the methods and the class invariant) discuss the following:

1. Are they valid when the class `Point` is used in a concurrent context, where multiple threads can simultaneously call the methods in `Point`? If so, explain why. If not, give a counter example.
2. If the specifications are not valid, how can the specifications and/or the program be adapted to make the specifications valid?

□

Correctness and performance of lazy initialization. Lazy initialisation is an important concept in object oriented programming, used for example in the *Singleton* design pattern. However, it can also be a source of errors in a multithreaded environment. Some problems with a simplistic (and wrong) implementation are discussed in the book (JCIP, 2.2.2, page 21-22).

Exercise 2-CP.3 *Learning goal:* optimization whilst maintaining correctness.

1. Implement a test for the class `LazyInitRace` of Listing 2.3 of JCIP, showing that the code snippet is indeed not thread safe.
2. Improve the class `LazyInitRace`, making sure that the class performs correctly in a multithreaded environment.
3. Optimize your class for speed, and document your improvements. It is possible to asymptotically obtain the same performance as the incorrect version.

□

Fine-grained locking. When implementing a concurrent data structure, there are several strategies you can use with regard to locking. The most simple and straightforward is to simply protect all accesses and changes of the data structure by a single lock. This does result in a correct program, but it can make the performance of your program less optimal than expected for a multithreaded program, as you essentially make all accesses to the data structure sequential.

This assignment deals with a different strategy: *fine-grained locking*. Simply put, if you want to reduce the congestion of a lock (which will be very high when you use a single lock) you could use multiple locks to spread the congestion to a more manageable level, improving the performance. In the case of a linked list, the most fine-grained solution is to protect every element with its own lock. When you only want to set, get, prepend or append values this approach is fairly straightfoward to implement. However, when the operations on the linked list also include e.g., insertions and deletions, only holding a single lock does not guarantee correctness any more. To solve this issue, the concept of *hand-over-hand* locking or *lock-coupling* was invented. A basic explanation of lock-coupling is given in the book (JCIP, 13.1.3, page 281-282).

Exercise 2-CP.4 *Learning goal:* complex locking policies.

1. *Implement* a linked list using lock-coupling, capable of (at least) insertions and deletions. Obviously, the list should behave correctly in a multithreaded environment, i.e., there should not be any deadlocks or invalid memory accesses.

   Your linked list class should (at least) implement the following interface:

```
interface List<T> {
    public void insert(int pos, T val);
    public void add(T val);
    public void remove(T item);
    public void delete(int pos);
    public int size();
    public String toString();
}
```

2. Report on the performance gains when compared with guarding the entire list by a single lock.
3. What are the issues that you encountered whilst developing your list implementation?

□

2-FP Functional Programming

To be added

2-CC Compiler Construction

2-CC.1 EC Section 3.3: Top-Down Parsing

Exercise 2-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.

Sentential form, parse tree, ambiguity, left/right recursion, recursive-descent parsing, LL(x), bottom-up parsing, LR(x).

□

Exercise 2-CC.2 Regard the following variation on the grammar on Page 12 of the book:

1	*Sentence*	$\rightarrow$	*Subject* `verb` *Object* `endmark`
2	*Subject*	$\rightarrow$	`noun`
3	*Subject*	$\rightarrow$	*Modifier Subject*
4	*Object*	$\rightarrow$	`noun`
5	*Object*	$\rightarrow$	*Modifier Object*
6	*Modifier*	$\rightarrow$	`adjective`
7	*Modifier*	$\rightarrow$	*Modifier Modifier*

After scanning, the sentence "all smart undergraduate students love compilers." gives rise to the token sequence

```
adjective adjective adjective noun verb noun endmark
```

Answer the following questions, where you may use the abbreviations:

S	*Sentence*	`n`	`noun`
U	*Subject*	`v`	`verb`
O	*Object*	`a`	`adjective`
M	*Modifier*	`e`	`endmark`

1. Give all leftmost derivations and all rightmost derivations of the above sentence, as well as the parse trees they generate.
2. Which parse tree best reflects the grammatical structure of the sentence?
3. How can you change the grammar so that it becomes unambiguous, and the only parse tree of the above sentence is the one you consider to be the best one?

□

Exercise 2-CC.3 Consider the following grammar (which is a variant of the grammar shown on p. 91 of EC):

1	*Stat*	$\rightarrow$	`assign`
2		$\mid$	`if expr then` *Stat ElsePart*
3	*ElsePart*	$\rightarrow$	`else` *Stat*
4		$\mid$	ε

Show, through a calculation of the FIRST-, FOLLOW- and FIRST$^+$-sets, that this grammar is not LL(1). In the calculation of the FIRST- and FOLLOW-sets, show the outcome after each iteration of the **while**-loops in Figs. 3.7 and 3.8, respectively. □

Exercise 2-CC.4 Make Exercise 3.4 from EC. *L* is the start symbol of the grammar. As in Exercise 2-CC.3, show the iterations in the calculation of FIRST and FOLLOW. *Note:* the terminals are the single `a`, `b` and `c`, *not* sequences such as `aba`. *Also note:* In EC (§3.3.1) it is explained how to turn left-recursive rules into right-recursive ones. □

2-CC.2 Programming the LL(1) algorithm

For the following questions, you have to do some programming. The lab files provided for this block contain the following general classes (in package `pp.block2.cc` and subpackages thereof):

- `Symbol`, `Term` and `NonTerm`: implementations of grammar symbols, further divided into terminals and nonterminals. `Symbol` also contains definitions of the special (terminal) symbols `EMPTY` amd `EOF`.
- `Rule`: a pair of a nonterminal (the rule's left hand side or LHS) and a list of symbols (the rule's right hand side or RHS). Note that if the RHS of a rule is empty, `Rule.getRHS()` will return an empty list, *not* a list containing `Symbol.EMPTY`.
- `Grammar`: a combination of a set of rules and a start symbol (a nonterminal). Note that the special terminals `EMPTY` and `EOF` will normally not occur in a `Grammar`-instance.
- `LLCalc` and `LLCalcTest`: an interface for the calculation of FIRST, FOLLOW and FIRST$^+$, and a JUNIT-test for your implementation of the same.
- `SymbolFactory`: a helper class that can extract token names from ANTLR grammars and build `Term`-instances from those. `SymbolFactory` is used in `LLCalcTest`.

The Javadoc provides further information. Moreover, there are two grammars included in the lab files:

- `Sentence.g4` and `If.g4`: two ANTLR lexer grammars providing the vocabulary of Exercise 2-CC.2 and Exercise 2-CC.3.

Exercise 2-CC.5 Write your own implementation of `LLCalc`, with a constructor that takes a `Grammar` as argument. Make sure you deal correctly with the special symbols `Symbol.EMPTY` and `Symbol.EOF`: those are *not* present in the constructed `Grammar`, you have to add and manipulate them explicitly in your `LLCalc`-implementation. Test your implementation using `LLCalcTest`.

Hint: In the calculation of the FIRST$^+$-set you have to compute FIRST(β) where β is the right hand side of a rule. Since this is also what happens in the **while**-loop of the FIRST-algorithm, it makes sense to define an auxiliary method for this. □

Exercise 2-CC.6 Extend `LLCalcTest` with tests for the grammars of Exercise 2-CC.3 and Exercise 2-CC.4.

1. Extend `Grammars` with analogous instances for the *If*-grammar of Exercise 2-CC.3 as well as the grammar of (your answer to) Exercise 2-CC.4. Use the *Sentence* grammar definition as a template. *Note:* in order to use the `SymbolFactory` class for the grammar of Exercise 2-CC.4, you have to create a corresponding ANTLR lexer grammar first.
2. Extend `LLCalcTest` with tests for these grammars, along the lines of:

```java
@Test
public void testIf() {
    Grammar g = Grammars.makeIf(); // to be defined (Ex. 2-CC.6)
    // Define the non-terminals
    NonTerm stat = g.getNonterminal("Stat");
    NonTerm elsePart = g.getNonterminal("ElsePart");
    // Define the terminals (take from the right lexer grammar!)
    Term ifT = g.getTerminal(If.IF);
    ... // (other terminals you need in the tests)
    Term eof = Symbol.EOF;
    Term empty = Symbol.EMPTY;
    LLCalc calc = createCalc(g);
```

```
        // FIRST
        Map<Symbol, Set<Term>> first = calc.getFirst();
        assertEquals(/* see 2-CC.3 */, first.get(stat));
        // (insert other tests, see 2-CC.6)
        // FOLLOW
        Map<NonTerm, Set<Term>> follow = calc.getFollow();
        assertEquals(/* see 2-CC.3 */, follow.get(stat));
        // (insert other tests)
        // FIRSTP
        Map<Rule, Set<Term>> firstp = calc.getFirstp();
        List<Rule> elsePartRules = g.getRules(elsePart);
        assertEquals(/* see */, firstp.get(elsePartRules.get(0)));
        // (insert other tests)
        assertFalse(calc.isLL1());
    }
```

(and similar for the grammar of Exercise 2-CC.4)

3. Confirm your own answers to Exercise 2-CC.3 and Exercise 2-CC.4 by running the extended test.

□

In Section 3.3.2 of EC, you can read about the principle of (hand-coded) recursive-descent parsers based on the LL(1) principle. In particular, if you have calculated the FIRST$^+$-set of the rules of your grammar, then the corresponding recursive-descent is really easy to write. (Actually, it becomes a bit harder if you also want to deal with errors in a usable fashion, in particular to make sure that your parsing doesn't just give up on the first error; however, we'll simply ignore this here. See Section 3.5.1 for a discussion.)

Exercise 2-CC.7 Among the lab files, you will also find `cc.block2.ll.SentenceParser`, which is a hand-written recursive-descent parser for the *Sentence*-grammar of Exercise 2-CC.2 — minus Rule 7, which (as you have seen) makes the grammar ambiguous. The class has a `main`-method that lets you try it out on texts you pass in as arguments.

1. Try out `SentenceParser` on a few sample inputs, both correct and wrong. Note that the lexer grammar `Sentence.g4` determines which words are actually recognised. in particular, confirm that the example sentence of Exercise 2-CC.2 gives rise to the parse tree corresponding to your answer to 2-CC.2.3.
2. Write a recursive-descent parser for the grammar of Exercise 2-CC.4, analogous to `SentenceParser`. (You should already have created the necessary lexer grammar in your solution to Exercise 2-CC.6.)
3. Test your solution to the previous subquestion on the input texts `abaa`, `cababcbca`, `bbcca` and `bbcba`.

□

Exercise 2-CC.8 You should now have grasped the principles of top-town, LL(1) parsing well enough to be able to write a table-driven parser. In Section 3.3.3 of EC (Figure 3.11) you can find a *non-recursive* algorithm for writing a table-driven parser. However, for this exercise you should build a *recursive* table-driven parser so that it can simultaneously produce a parse tree. The lab-file `pp.block2.cc.ll.GenericLLParser` gives a skeleton for you to fill in.

1. Program `GenericLLParser`.
2. Make sure that `pp.block2.cc.test.GenericLLParserTest` shows no errors.
3. Extend `GenericLLParserTest` with a method to test `GenericLLParser` by comparing the grammar of Exercise 2-CC.4 against the hand-written parser of Exercise 2-CC.7, analogous to the test for the *Sentence*-grammar in `testSentence()`.

Note that the implementation of the LL(1)-parse table (Figs. 3.11(b) and 3.12 of EC, see p. 112–113) foreseen in this file is of type `Map<NonTerm, List<Rule>>`, which maps every non-terminal to a list of rules indexed by token type. The token types are the numbers corresponding to the tokens in the ANTLR-generated lexer. □

2-CC.3 ANTLR parser grammars

In `pp.block2.cc.antlr.Sentence.g4` you will find a full grammar specification (not just a lexer) for the *Sentence*-grammar of Exercise 2-CC.2. (It is actually more elegant to *import* a lexer grammar into a full grammar, rather than just copy/pasting the whole thing as done here; but in the context of this course, the chosen package structure gets in the way.) The first few lines of the grammar are:

```
grammar Sentence;

@header{package pp.block2.cc.antlr;}

/** Full sentence: the start symbol of the grammar. */
sentence: subject VERB object ENDMARK;
/** Grammatical subject in a sentence. */
subject: modifier subject | NOUN;
/** Grammatical object in a sentence. */
object: modifier object | NOUN;
/** Modifier in an object or subject. */
modifier: ADJECTIVE;
```

The following should be noted:

- The first line now reads **`grammar`** rather than **`lexer grammar`**, reflecting the fact that this contains both parser and scanner (i.e., lexer) rules.
- The rules `sentence` etc. have essentially the same syntax as the lexer rules; in fact, the most prominent difference is that the names of the nonterminals start with a lowercase letter.

If you now generate the corresponding recogniser, you will get four JAVA files instead of just one:

- `SentenceListener`: an interface for listeners to the parser. We'll see below how to use listeners.
- `SentenceBaseListener`: a skeleton implementation of the above, with empty listener methods.
- `SentenceLexer`: this equals the lexer class generated for the lexer grammar.
- `SentenceParser`: this class inherits from `org.antlr.v4.runtime.Parser`, which in turn offers functionality to add and remove `SentenceListener`s.

Exercise 2-CC.9 Generate the above files from `pp.block2.cc.antlr.Sentence.g4`, and run `SentenceUsage`. Study the class and make sure you understand what goes on. What happens if you parse an incorrect text? □

The best way to use ANTLR parse trees such as returned by `SentenceParser.sentence()` is to *walk* the tree using a tree listener; specifically, a `SentenceListener`. See `https://theantlrguy.atlassian.net/wiki/display/ANTLR4/Parse+Tree+Listeners` and elsewhere on the web to find more information on how to program and invoke ANTLR tree listeners. Furthermore, `pp.block2.cc.antlr.SentenceCounter` contains an example.

Exercise 2-CC.10 Program `SentenceConverter` to transform ANTLR parse trees of the `Sentence`-language to `pp.block2.cc.AST` instances.

1. Make sure you can run `pp.block2.cc.antlr.SentenceCounter` and understand how the code works.
2. Extend `pp.block2.cc.antlr.SentenceConverter` given in the lab files.
3. Test your solution using the provided `SentenceConverterTest` class. □

ANTLR grammars are in fact quite a bit more powerful than LL(1)-grammars. Among other things, direct left-recursion is automatically factored out. This means that it is often possible to write rules that directly express the intended tree structure, as for instance in the case of operator precedence (also treated extensively in Chapter 3 of EC).

For the following question, there are some special features of ANTLR that come in quite handily.

- The alternatives of a rule are processed in the order given in the grammar. Thus, the first alternative is tried out first. This makes it convenient to program operator precedence.

- You can *name* the top-level alternatives of a rule by inserting a tag of the form `# myName` into the grammar. This will result in modified visitor methods that are called precisely when that alternative is visited, rather than every time the whole rule is visited. For instance, in the `Sentence` grammar above, you could write the rule for `subject` as

  ```
  subject : modifier subject # modSubject
          | noun             # simpleSubject
          ;
  ```

  resulting in visitor methods `[enter/exit]ModSubject` and `[enter/exit]SimpleSubject`, *replacing* the previous `[enter/exit]Subject`.

- You can explicitly set the associativity of a top-level alternative by inserting the directive `<assoc=right>` directly in front of the alternative. For instance, in `Sentence.g4` above, you could change the `modifier` rule to

  ```
  modifier : <assoc=right> modifier ',' modifier
           | adjective
           ;
  ```

  resulting in a non-ambiguous grammar in which adjectives are interpreted in a right-associative manner.

Exercise 2-CC.11 Try out the features explained above (on a *copy* of the `Sentence` grammar, so you don't ruin your solutions to the exercises above!)

1. Change the rule for `subject` in the way described above, and study the resulting `SentenceListener` interface. What parameters do `enterModSubject` and `enterSimpleSubject` etc. get, and what functionality do those parameters offer?
2. Change the rule for `modifier` in the way described above, and try parsing the sentence of Exercise 2-CC.2 (with commas inserted between the adjectives). If you look again at your answer to that exercise, which parse tree does the one now generated by ANTLR correspond to? If you leave out the **assoc**-directive (but leave in the left-recursive alternative itself), which parse tree do you then get? (Use the "Parse Tree View" of your Eclipse plugin as a fast way to check this.) □

Exercise 2-CC.12 Design a language for arithmetic expression that includes addition (e.g., `1+2`), subtraction (e.g., `2-3`), multiplication (e.g., `2*3`), negation (e.g., `--2` should stand for double negation, yielding 2) and exponentiation (e.g., `2^3`), in increasing order of precedence. Addition, subtraction and multiplication are left-associative (for instance, `1-2-3` stands for `(1-2)-3` and not `1-(2-3)`) but exponentiation is right-associative (for instance, `2^3^4` stands for `2^(3^4)` and not `(2^3)^4`). Of course, the language should also provide parentheses and (natural) numbers.

1. Write an ANTLR grammar for this arithmetic expression language.
2. Program a `Calculator` as a tree visitor that can take an expression in this language and compute the outcome, based on `BigInteger` values.
3. Write a test for your class where you demonstrate all features of the expression language. □