Block 2

Block 2

2-OV Overview

2-OV.1 Contents of This Block

CP This block will quickly recapitulate the basicx constructs to write concurrent Java programs, as discussed in Module 1.2 (Software systems). Further, we will discuss the meaning of thread safety, different ways to share data between threads, and how to test concurrent programs.

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:

- 4 hours self-study for the CP exercises;
- 2 hours self-study to study the CC material.

2-OV.4 Materials for this Block

- **CP**, from JCIP: Chapters 1- 3 and 12.
- **CC**, from EC: Sections 3.1–3.3 (intensively), Sections 3.5–3.6 (superficially).

2-CP Concurrent Programming

During this week, you are asked to solve a collection of basic concurrent programming 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 and discuss your solutions with your fellow students.

The objective of this first set of exercises is twofold. Firstly, we want you to brush up your Java knowledge with some simple multi-threaded Java applications. Secondly, you will discover that concurrency bugs are not just theory, but easily happen in practice.

Note that several exercises ask you to show that something is not thread safe. On Blackboard we provide a simple test framework, which you can reuse, such that you do not have to write the boilerplate code, e.g., for spawning and joining threads, every time.

Unsafe Sequence. The goal of the first exercise is to get more insight in the possible behaviours a concurrent program can have. You are asked to implement a class that is not thread-safe, and to demonstrate that it is indeed not thread-safe by analysing the different behaviours.

Exercise 1-CP.1 *Learning goal:* testing a concurrent application, identifying non-thread-safe behaviour.

1. Take the non-thread-safe sequence generator from Listing 1.1 (JCIP, page 6). Add a test that allows you to demonstrate that it is indeed *not* thread-safe. Due to the randomness of the thread interleaving, different runs will probably yield different results.
2. Take the output of one of your test runs and show where thread safety is violated. Give an interleaving that could have led to the behaviour you observed.
3. Develop different alternatives to make the class thread safe and show that these implementations are indeed working properly.

Hint: Even though the sequence generator is not thread-safe, your *test class* should be thread-safe in order to get valid results. □

Producer-Consumer Pipelines. Queues in their various different forms are essential building blocks for multithreaded processing pipelines. In order to achieve maximal throughput it is important that the queue allows for multiple producers to add messages to the queue, and for multiple consumers to process the messages, each running possibly in its own thread. Standard examples of queues are:

- stack (FILO, First In Last Out)
- buffer (FIFO, First In First Out)
- priority queue (the message with the highest priority gets consumed first)

Exercise 1-CP.2 *Learning goal:* ensuring thread-safety.

In this exercise you are asked to investigate how such a producer-consumer pipeline can be used safely in a concurrent environment. Create your queue (or buffer) adhering to the following simple interface.

```
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. Implement a queue, based on a linked list (i.e., a node contains a reference to the next node) without worrying about concurrency considerations.

2. Implement a test class and document the events that cause your data structure to fail for multiple consumers and producers.
   Hint: Again, make sure your test class is thread safe in order to obtain valid results.
3. Change your implementation to make it thread safe and show that your queue passes the tests from assignment 2.
4. Are there ways to further improve your implementation, using finer-grained concurrency, to improve concurrency? If possible, adapt your implementation accordingly.
5. Due to the randomness of thread interleaving it is *impossible* to prove the correctness of the synchronisation by merely running a test. Provide an informal proof why your synchronisation method achieves correctness.

□

Thread Safety. It is often hard to see from documentation whether a class is thread safe and under what conditions.

Exercise 1-CP.3 *Learning goal:* identifying thread-safety-related problems.

For this exercise we will analyse the class `java.util.Timer`. Before proceeding, carefully read the javadoc documentation of `java.util.Timer`.

1. Create a program which has a *single* `Timer` object with n `TimerTask`s associated. Each of these `TimerTask` objects should trigger at the exact same moment t_0. Use `Timer`'s `schedule` method to start the tasks at the same moment t_0 (or after a certain delay). In the `run` method of the `TimerTask` objects you should increment a variable which is shared by all `TimerTasks` objects. After all tasks have finished, check the number of updates to the variable.
2. Create a program which has a n `Timer` objects which trigger at the exact same moment t_0. Each of theses n `Timer` objects gets a single `TimerTask` object which is triggered at t_0. In the `run` method of the `TimerTask` objects you should increment a variable which is shared by all `TimerTask` objects. Again, after all tasks have finished, check the number of updates to the variable.
3. What do you observe? Under what conditions does everything work out and under what conditions does it go wrong? Based on your observations, what do you conclude about the implementation of `java.util.Timer`?

□

Exercise 1-CP.4 *Learning goal:* identifying thread-safety-related problems.

Paragraph 4.5 of JCIP (page 75) mentions that `java.text.SimpleDateFormat` is not thread safe (and so does the javadoc of `SimpleDateFormat`). Show that this is indeed the case.
Hint: Breaking things is usually easier than guaranteeing their correctness. Create some threads which share a `java.text.SimpleDateFormat` object. Let each thread update the shared object to random (but correct!) dates using the method `parse`. □

Assertions and Concurrency. Concurrent execution also has an impact on the claims that you can make about the behaviour of a program. As discussed in Section 3.5.2 of JCIP, concurrency can make assertions invalid, even though from a sequential point of view they would be considered obviously valid.

JML specifications also can be considered as assertions: every precondition translates to an assert at the beginning of the method; every postcondition translates to an assert at the end of the method.

Exercise 1-CP.5 *Learning goal:* understanding impact of concurrency on reasoning about the behaviour of an application.

Consider the JML-annotated class `Point`.

```
public class Point {
  /*@ spec_public */ private int x;
  /*@ spec_public */ private int y;

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

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

  /*@
    requires n >= 0;
    ensures getX() == \old(getX()) + n;
   */
  public void moveX(int n) {
      x = x + n;
  }

  /*@
    requires n >= 0;
    ensures getY() == \old(getY()) + n;
   */
  public void moveY(int n) {
      y = y + n;
  }
}
```

This class `Point` is used in a class `RandomDrift`, which moves the point up and to the right with random steps.

```
public class RandomDrift extends Thread {
    private Point p;

    public RandomDrift(Point p) {
        this.p = p;
    }

    @Override
    public void run() {
        while (true) {
            int n = (int) (Math.random() * 10);
            p.moveX(n);
            int m = (int) (Math.random() * 10);
            p.moveY(m);
        }
    }
}
```

1. Class `RandomPoint` starts two `RandomDrift` threads, using the same instance of `Point`. Describe the state space of this program.
2. Discuss where JML specifications might become violated because of concurrency aspects. Show an example execution that violates the specifications.
3. Suppose the methods `moveX` and `moveY` in `Point` become synchronized. Would this solve the problem? Argue why, or why not.

□

2-FP Functional Programming

The material for FP is provided separately on BLACKBOARD.

2-CC Compiler Construction

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

Exercise 2-CC.1 Consider the following grammar (which is a variant of the grammar shown on p. 91 of EC):

1	*Stat*	→	`assign`
2		|	`if expr then` *Stat ElsePart*
3	*ElsePart*	→	`else` *Stat*
4		|	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.2 Make Exercise 3.4 from EC. *L* is the start symbol of the grammar. As in Exercise 2-CC.1, 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 1-CC.16 and Exercise 2-CC.1.

Exercise 2-CC.3 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.4 Extend `LLCalcTest` with tests for the grammars of Exercise 2-CC.1 and Exercise 2-CC.2.

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

```
Grammar ifG = Grammars.makeIf(); // to be defined (Ex. 2-CC.4.1)
// Define the non-terminals
NonTerm stat = ifG.getNonterminal("Stat");
NonTerm elsePart = ifG.getNonterminal("ElsePart");
// Define the terminals (take from the right lexer grammar!)
Term ifT = ifG.getTerminal(If.IF);
... // (other terminals you need in the tests)
Term eof = Symbol.EOF;
Term empty = Symbol.EMPTY;
LLCalc ifLL = createCalc(ifG);

@Test
public void testIfFirst() {
    Map<Symbol, Set<Term>> first = ifLL.getFirst();
    assertEquals(/* see 2-CC.1 */, first.get(stat));
    // (insert other tests)
}

@Test
public void testIfFollow() {
    Map<NonTerm, Set<Term>> follow = ifLL.getFollow();
    assertEquals(/* see 2-CC.1 */, follow.get(stat));
    // (insert other tests)
}

@Test
public void testIfFirstPlus() {
    Map<Rule, Set<Term>> firstp = ifLL.getFirstp();
    List<Rule> elseRules = ifG.getRules(elsePart);
    assertEquals(/* see 2-CC.1 */, firstp.get(elseRules.get(0)));
    // (insert other tests)
}

@Test
public void testIfLL1() {
    assertFalse(ifLL.isLL1());
}
```

   (and similar for the grammar of Exercise 2-CC.2)
3. Confirm your own answers to Exercise 2-CC.1 and Exercise 2-CC.2 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.5 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 1-CC.16 — 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 1-CC.16 gives rise to the parse tree corresponding to your answer to 1-CC.16.3.
2. Write a recursive-descent parser for the grammar of Exercise 2-CC.2, analogous to `SentenceParser`. (You should already have created the necessary lexer grammar in your solution to Exercise 2-CC.4.)
3. Test your solution to the previous subquestion on the input texts `abaa`, `cababcbca`, `bbcca` and `bbcba`.

□

Exercise 2-CC.6 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.2 against the hand-written parser of Exercise 2-CC.5, 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 1-CC.16. (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:

```
1  grammar Sentence;
2
3  @header{package pp.block2.cc.antlr;}
4
5  /** Full sentence: the start symbol of the grammar. */
6  sentence: subject VERB object ENDMARK;
7  /** Grammatical subject in a sentence. */
8  subject: modifier subject | NOUN;
9  /** Grammatical object in a sentence. */
10 object: modifier object | NOUN;
11 /** Modifier in an object or subject. */
12 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.7 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://github.com/antlr/antlr4/blob/master/doc/listeners.md` 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.8 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.9 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 1-CC.16 (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.10 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. □