Block 3

Block 3

3-OV Overview

3-OV.1 Contents of This Block

CP This block will discuss synchronisation using locks (both intrinsic and explicit), condition variables, and other synchronizers.

FP From this block onward, several non-trivial applications will be discussed and practiced. In block 3, the main issue will be parsing.

CC This block will address the "elaboration" phase of the first stage of compilation. This involves especially type and scope checking/inference. As solutions, you will learn about attribute grammars, syntax-directed translation and symbol tables.

3-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;
- 4 hours self-study to complete the FP lab exercises.
- 2 hours self-study to read through the CC material and complete the lab exercises.

3-OV.3 Materials for this Block

- **CP**, from JCIP: Chapters 4, 5, 13, and 14.1-14.4.
- **CC**, from EC: Chapter 4 and Section 5.5.

3-CP Concurrent Programming

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.

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

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

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

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

    // every desk employee should run as a separate thread
    @Override
    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 (note, there are more than three errors!). □

Exercise 2-CP.2 *Learning goal:* understanding that the preservation of a 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 = 0;
    /*@ spec_public */private int y = 1;
    //@ public invariant x != y;
    private final Object lockX = new Object();
    private final 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?
3. What happens if we change the order of the locks in `moveY`: first acquire `lockY` and subsequently acquire `lockX`?

□

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.3 *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?

□

Custom synchronizers. The Java concurrency library contains quite a few different and useful *synchronizers*: a wide variety of locks and semaphores. However, in order to really understand the inner workings of a synchronizer it is instructive to implement one yourself. The first binary mutual-exclusion (*mutex*) algorithm which was proven to be correct was Dekker's algorithm.

Exercise 2-CP.4 Consider the following `BasicLock` interface which can be used to protect a critical section. The argument `thread_nr` makes it easier to implement the algorithms of this exercise.

```
public interface BasicLock {
    /**
     * Aqcuires the lock.
     * thread_nr is the number of the requesting thread, thread_nr == 0|1
     */
    public void lock(int thread_nr);

    /**
     * Releases the lock.
     * thread_nr is the number of the releasing thread, thread_nr == 0|1
     */
    public void unlock(int thread_nr);
}
```

1. Look up a pseudocode version of Dekker's algorithm and use it to make an implementation of `BasicLock`. Write a simple test for your mutex class and use it to verify your implementation. For

example, you could use the `UnsafeSequence` class of block 1, Exercise 1, and show that your mutex implementation protects an unsafe variable in a thread-safe way.

Hint: Instead of using two `volatile` boolean variables to express the desire to enter the critical section you could consider to use a `AtomicIntegerArray`.

2. Create a new mutex class, based on the *compare-and-set* instruction from the `AtomicInteger` class. Start out with a non-reentrant version, suitable for only two threads. Re-use the test harness of Dekker's implementation and validate that your *compare-and-set* mutex works correctly.
3. Expand the *compare-and-set* mutex implementation to accomodate reentering. If a thread holds the lock and requests it again this will be allowed; a non-reentrant lock would cause a deadlock in this scenario.
4. Compare the various synchronizers: Dekker's mutex algorithm, the *compare-and-set* mutex, and the `ReentrantLock` from the Java library.

□

Barriers and map-reduce. Barriers (or fences), as the name implies, are used to separate your program into sections; execution is only allowed to continue whenever *all* threads in the section have finished that section. A simple example in which this is useful is adding all numbers in a certain array. A single-threaded solution would use a single thread to loop over all elements. However, when using a barrier, you can instruct thread `1` to sum the first half of the array and thread `2` to sum the second half of the array, almost halving the execution time, though one still needs to add the two sub-sums to obtain the final answer.

This simple example is a course-grained concurrent version of the approach known as *map-reduce*. The first step (*map*) is to apply a function to all elements of the collection; in this example, the two sub-arrays. The executions of these functions are independent. The second step (*reduce*) is to apply a function which merges the results of the map-operation into the final answer.

A more realistic example of map-reduce is a possible implementation of a web analyzer, e.g., Google's PAGERANK. The input of this algorithm is a very large set of HTML files, found by a webcrawler. These are processed individually in order to extract keywords, find hyperlinks and anything else which might be useful in later analysis. This is the map-stage. Next, this mapped data is fed to the reducer where it gets combined to obtain useful information.

Note: you might recognize the term `map` from its origin in functional programming.

Exercise 2-CP.5

- Create a minimal *map-reduce* framework in Java using barriers. The idea is to develop an abstract `MapReduceBase` class, which has two abstract methods `map` and `reduce`, which will be implemented by subclasses of `MapReduceBase`. The class `MapReduceBase` is responsible for spawning the threads that perform the `map` and combining the results with `reduce`.
- As always: devise a test, and apply it to make sure that your framework works as expected.

□

3-FP Functional Programming

The material for FP is provided separately on BLACKBOARD.

3-CC Compiler Construction

3-CC.1 EC Chapter 4: Type inference and attribute grammars

Exercise 3-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.

Operator overloading, type inference, synthesized resp. inherited attribute, syntax-directed translation.

□

Exercise 3-CC.2 Answer Review Question 1 of Section 4.2 (p. 181 of EC) for JAVA. Explicitly compare the types in JAVA with each of the categories discussed in Section 4.2.2 (base types, compound types etc.). □

Consider an expression language with types `Num` (numbers), `Bool` (booleans) and `Str` (strings), and the following overloaded operators, in decreasing order of precedence:

- Hat (`^`): either exponentiation of numbers (e.g., `3^2`, which equals 9) or duplication of strings (e.g., `"ab"^3`, which equals `"ababab"`)
- Plus (`+`): either addition of numbers (e.g., `2+3`, equalling `5`) or concatenation of strings (e.g., `"ab"+"ac"`, equalling `"abac"`) or disjunction ("or") of booleans (e.g., **`true`**`+`**`false`**, equalling **`true`**)
- Equals (`=`): equality between any pair of values of the same type, always yielding a boolean.

This gives rise to the following grammar (never mind that it is not LL(1), that is not important right now):

1	$T \rightarrow T$ `^` T	6	$T \rightarrow$ `num`
2	| T `+` T	7	| `bool`
4	| T `=` T	8	| `str`
5	| `(` T `)`		

Exercise 3-CC.3 Answer the following questions for the language sketched above.

1. For each of the operators, sketch tables like the one in Fig. 4.1 of EC showing the result type of an expression given the type of its operands.
2. Give attribution rules for the type inference of the grammar, in the style of Figs. 4.5 and especially 4.7. (Note: the rules should express type inference, not computation of the result!)
3. Are your attributes synthesized or inherited?

□

To operationalize the attribute rules, we turn again to ANTLR. As an example, consider the following grammar, which defines yet another type of simple expression, with attribute rules that calculate the *value* of the expression that has just been parsed.

1	$E_0 \rightarrow E_2$ `*` E_1	E_0`.val` $\leftarrow E_1$`.val` $*$ E_2`.val`
2	| E_1 `+` E_2	E_0`.val` $\leftarrow E_1$`.val` $+$ E_2`.val`
3	| `(` E_1 `)`	E_0`.val` $\leftarrow E_1$`.val`
4	| `number`	E_0`.val` $\leftarrow$ `number.val`

`pp/block3/cc/antlr/Calc.g4` contains an ANTLR grammar for this language, and `CalcAttr.g4` is a variant of this grammar where the attribute rules have been implemented using explicit, built-in *actions*. (In reality this is an imperative solution, more like the ad hoc syntax-directed rules of Section 4.4.) Note the use of the method `getValue` in the rule for `NUMBER`: this is a user-defined method whose declaration is given in the **`@members`** block of the grammar. This piece of user-defined code is included in the `CalcAttrParser` class generated by ANTLR from the grammar definition.

On the other hand, the class `pp.block3.cc.antlr.Calculator` in combination with the grammar `Calc.g4` in the same package shows an alternative solution where the attribute rules are implemented through a *tree listener*, as already encountered in Block 2. In particular, the `vals`-field, declared to be of type `ParseTreeProperty<Integer>`, implements the required attribute, *not* by including an **`int`**-field in every node of the parse tree but through a *map* from parse tree nodes to `Integer`s.

The class `pp.block3.cc.test.CalcTest.java` is a JUNIT test for both solutions.

Exercise 3-CC.4 Study the action-based and listener-based solutions for the attribute rules.

1. Add a unary minus operator (i.e., negation, as in $1 + -2$) to both `Calc.g4` and `CalcAttr.g4` and extend the attribute rules and the `Calculator` class to cover this new case. Also extend the test (and make sure it runs).
2. What are advantages of the action-based solution, and what are advantages of the listener-based solution?
3. The subrule for `NUMBER` in `CalcAttr.g4` not only has an action *after* the evaluation of the right hand side, but also *before* evaluation (the `println`-statement). What happens if you also add such `println`-statements in front of the other subrules? Can you explain this effect?
4. What is the relation of the tree listener methods to the concept of synthesized versus inherited attributes? □

Exercise 3-CC.5 Give two different ANTLR grammars for the language of Exercise 3-CC.3 and implement your attribute rules for type inference in the following ways:

1. Define a grammar analogous to `CalcAttr.g4`, including direct actions for type inference. Use the provided **`enum`** type `pp.block3.cc.antlr.Type` to represent the inferred types.
2. Define a grammar analogous `Calc.g4` with a corresponding listener.
3. Program a JUNIT test for your grammars, analogous to `CalcTest` above, in which you show that type inference works correctly for all operators (including correctly flagging type errors).

□

3-CC.2 Listeners and symbol tables

Symbol tables The grammar `pp/block3/cc/symbol/DeclUse.g4` (provided in the lab files) contains the following parser rules:

```
program : '(' series ')' ;
series  : unit* ;
unit    : decl | use | '(' series ')' ;
decl    : 'D:' ID ;
use     : 'U:' ID ;
```

This defines a simple language with nested scopes, in which variables are *declared* (`D:var`) and *used* (`U:var`). An example program is

```
(D:aap (U:aap D:noot D:aap (U:noot) (D:noot U:noot)) U:aap)
```

The policies for declaration and use are:

- An identifier may only be used if it has been declared before, in the same scope or an enclosing one.
- An identifier may not be redeclared in the same scope.
- An identifier *may* be redeclared in an inner scope. This inner declaration temporally "overwrites" the previous (outer) one.

To keep track of declarations and uses, we need a data structure called a *symbol table*, which keeps track of nested scopes. An interface for this data structure is provided in `pp.block3.cc.symbol.SymbolTable`, and a test for it in `pp.block3.cc.test.SymbolTableTest`.

Exercise 3-CC.6 Program your own implementation of the `SymbolTable` interface, and make sure it passes the test. □

To solve the next exercise, you have to read your input from a file rather than a string. This is actually straightforward: instead of `CharStream.fromString`, use the static method `CharStream.fromFileName` to create the character stream to pass into the lexer.

Take care! If you import an ANTLR type into your JAVA code, there is often a choice between equally named classes from ANTLR v3 and ANTLR v4. In that case, you *always* have to choose the v4-variant. You can distinguish it by the fact that there is a `v4` somewhere in the package name.

Exercise 3-CC.7 Program a tree listener that checks the declare/use policy outlined above, for a given "program" in the `DeclUse` language, using your `SymbolTable`. Your solution should:

- Collect all errors it finds in a list of understandable error messages *including line and column number of the token that caused the error.* This information is available through methods `Token.getLine()` and `Token.getCharPositionInLine()`. (The class `ParserRuleContext`, which all `...Context` classes extend, has a number of methods to retrieve the tokens from which a given parse (sub)tree was constructed.)
- Be able to return this error list after having walked a parse tree.

Provide some sample (correct and incorrect) programs and a JUNIT test that shows that your solution gives the right answers. □

From LaTeX to HTML. LaTeX is a powerful typesetting language, which is used throughout the scientific world. It is especially useful for texts containing a lot of mathematics and formulas. Typesetting a document in LaTeX has many similarities to programming.

An example LaTeX feature is the `tabular`-environment. The `tabular`-environment can be used to specify and format tables. An example table specification and formatting is the following:

```
% An example to test the Tabular application.
\begin{tabular}{lcr}
  Aap  & Noot & Mies \\
  Wim  & Zus  & Jet  \\
  1    & 2    & 3    \\
  Teun & Vuur & Gijs \\
\end{tabular}
```

Amongst others, this example table can also be found in the lab files in `tabular-1.tex`. A basic description of the `tabular`-environment is as follows.

- The following characters are reserved keywords in LaTeX: `\ { } $ & # ^ _ ~ %`
- Whitespace is *not* ignored in LaTeX; see below for the treatment of whitespace.
- The symbol `%` starts a comment line; the comment runs to (and includes) the end of the line (similar to JAVA comments starting with `//`). Comments are ignored.
- The start and end of a `tabular`-environment are given by the strings `\begin{tabular}` and `\end{tabular}`. The string `\begin{tabular}` expects one argument between curly braces, which is a non-empty string of characters `l` (left), `c` (centered) and `r` (right) specifying column alignments. In this exercise, we will disregard the argument.
- The entries of a `tabular` are specified row by row. The end of a row is specified with a double backslash: '`\\`'.
- Each row consists of entries separated with an ampersand: '`&`'.
- All tokens discussed above (table *start*, table *end*, and the *row* and *column* separators) may be preceded and followed by whitespace characters; these are considered to be part of the token.
- Entries consist of a possibly empty sequence of non-special characters, optionally separated by (but not starting or ending with) spaces.

Exercise 3-CC.8 The first part of writing a `tabular` compiler is to create a parser:

1. Given the listed description above, what are suitable tokens for scanning LaTeX tabular environments?
2. Write an ANTLR parser for `tabular` environments. Test your grammar by making sure that the provided auxiliary test files `tabular-[1...5].tex` parse correctly. □

You will now write a tree listener that transforms a `tabular` parse tree into an HTML table. The grammar of HTML table documents is specified in Figure 3.1. For instance, the specification of `tabular-1.tex` above should be translated to

```
<html>
<body>
<table border="1">
```

Doc	→	`<html> <body>` *Table* `</body> </html>`
Table	→	`<table border = "1">` *Row** `</table>`
Row	→	`<tr>` *Entry** `</tr>`
Entry	→	`<td>` *Text* `</td>`
Text	→	(any character except < and `&`)

Figure 3.1: Simplified grammar of HTML table document.

```
<tr>
  <td>Aap</td>
  <td>Noot</td>
  <td>Mies</td>
</tr>
<tr>
  <td>Wim</td>
  <td>Zus</td>
  <td>Jet</td>
</tr>
<tr>
  <td>1</td>
  <td>2</td>
  <td>3</td>
</tr>
<tr>
  <td>Teun</td>
  <td>Vuur</td>
  <td>Gijs</td>
</tr>
</table>
</body>
</html>
```

Before starting the output generation phase, we have to make sure the parsing phase encountered no errors.

ANTLR organises error reporting with a listener of type `org.antlr.v4.runtime.BaseErrorListener`. By default, every lexer and parser that ANTLR generates has an implementation of such a listener that only prints errors to `stderr`. To get more control over the error reporting process, you may remove any previous error listeners, and add your own implementation of `BaseErrorListener`.

```
Recognizer.removeErrorListeners()        // remove default reporting to stderr
Recognizer.addErrorListener(myListener)  // add your own error listener
```

Finally, it's important to realise that ANTLR's scanning phase is interwoven with the parsing phase. Thus in general, you can not report all lexer errors before all parser errors.

Exercise 3-CC.9 The second part of writing a `tabular` compiler is to create an HTML output generator:

1. Investigate the method `syntaxError` of `BaseErrorListener`. What parameters does it have?
2. Write your own error listener that formats the error message in the same way as the default reporter (including line and column number and an indication of the offending symbol), but saves the errors in a list rather than sending them directly to `stderr`. Add the error listener to the lexer and parser as described above.
3. Build a tree listener for tabular parse trees that writes the corresponding HTML table to a file.
4. Add to your tree listener a `main`-method that accepts a file name, reads and parses it, and only calls the HTML-conversion if the parse tree is error free.
5. Test your result by trying it on the provided lab files `tabular-[1...5].tex`, and visually inspecting the resulting HTML in a browser. □