Block 3

Block 3

3-OV Overview

3-OV.1 Contents of This Block

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

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:

- 6 hours self-study to prepare the CP 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 10, 11 and 12.
- **CC**, from EC: Chapter 4 and Section 5.5.

3-CP Concurrent Programming

Also for the third exercise session, you should try to solve a collection of self-study exercises. During the plenary exercise session, you will be asked to present your solutions, and the different solutions will be discussed in the group.

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 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. As mentioned in Block 1, 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 Recall the `Queue` interface from Block 1.

```java
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 1. 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 1 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.

□

Debugging a web client. Web browsers are easy targets for optimization by means of added concurrency. You can have separate threads to handle for example:

- the different downloads,
- rendering the pages, and
- displaying interactive content.

This works fine, as all of these activities are quite detached from each other.

Exercise 3-CP.3 The `zip`-file on Blackboard contains a very naive web client, e.g., it only performs requests (in plain TCP packets, no HTTP, it uses caching to reduce the amount of network traffic required, and it gathers some logging information about the performance. However, this web client contains several concurrency bugs.

- Find as many bugs as you can in the supplied Java web client, and explain why they are bugs.
- Report on the events that could lead to failure of the program.

- Fix the bugs whilst maintaining the performance, and document your fixes.

Note, making the entire application single threaded would of course fix the concurrency bugs, but would defeat the purpose of this assignment. □

3-FP Functional Programming

See 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	$\mid\ T$ `+` T	7	$\mid$ `bool`
4	$\mid\ T$ `=` T	8	$\mid$ `str`
5	$\mid$ `(` 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	$E_0 \to E_2$ * 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 file `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 grammars 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.

□

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: the class `AntlrInputStream` has a constructor that takes a *Reader* as parameter; all you have to do is to create a `FileReader` from the input file and pass that as an argument to the constructor.

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 Give 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()`.
- 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: '`\\`'.

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

Figure 3.1: Simplified grammar of HTML table document.

- 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">
<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 the 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. □