Block 4

Block 4

4-OV Overview

4-OV.1 Contents of This Block

CP This block will discuss the concurrency collections, different synchronisers, and condition variables.

CC This block will discuss several forms of intermediate representation: Contrl Flow Graphs, Dependence Graphs, ILOC with Static Single Assignment, and symbol tables.

4-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.

4-OV.3 Materials for this Block

- **CP**, from JCIP: Chapters 5 and 14, except 14.5 and 14.6.
- **CC**, from EC: Chapter 5.

4-CP Concurrent Programming

For this 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.

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 4-CP.1 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 `AtomicBoolean` or `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.

□

Copy-on-write. There are many scenarios in which the ratio between reads and writes of a data structure skews heavily in favour of reading. In these cases it might be advantageous to use a *copy-on-write* data structure. The principle of operation is that every *read* of information can be done directly. However, when you want to *write* to the structure the process is a bit more complex:

- Acquire the write lock.
- Make a *deep copy* of the main data structure into thread-local memory. This means that you also need to copy all other objects referenced by the data structure.
- Perform the write operation on the local object. For some structures (e.g., arrays) this might only modify a single object. For other, more complicated objects, more work might be involved. For example, a single write in a balanced tree might trigger a rebalance, thereby possibly affecting the entire tree.
- Replace the main structure by the updated version (i.e., with the write operation applied).
- Release the write lock.

Exercise 4-CP.2

1. Implement a *copy-on-write* list. Implement at least the methods `add`, `set` and `get`.

2. Compare the performance of your implementation with a `Collections.synchronizedList` on the basis of an `ArrayList`. Compare the performance of your implementation with the class `CopyOnWriteList` from the Java concurrency library. Can you explain your results?
3. Why is there no `CopyOnWriteMap` present in 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 4-CP.3

- 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.

□

4-FP Functional Programming

See BLACKBOARD.

4-CC Compiler Construction

4-CC.1 Control flow and dependency graphs

Exercise 4-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.

> *AST (esp. difference with parse tree), DAG, basic block, CFG (two meanings!), dependence graph, call graph, SSA, symbol table*

□

Consider the program fragment in JAVA in Figure 4.1(a), which finds the maximum element in an array `a` of non-negative integers. The control flow graph and dependency graph of this fragment are depicted as (b) and (c). The numbers in the figure correspond to the line numbers of the fragment. From the control flow

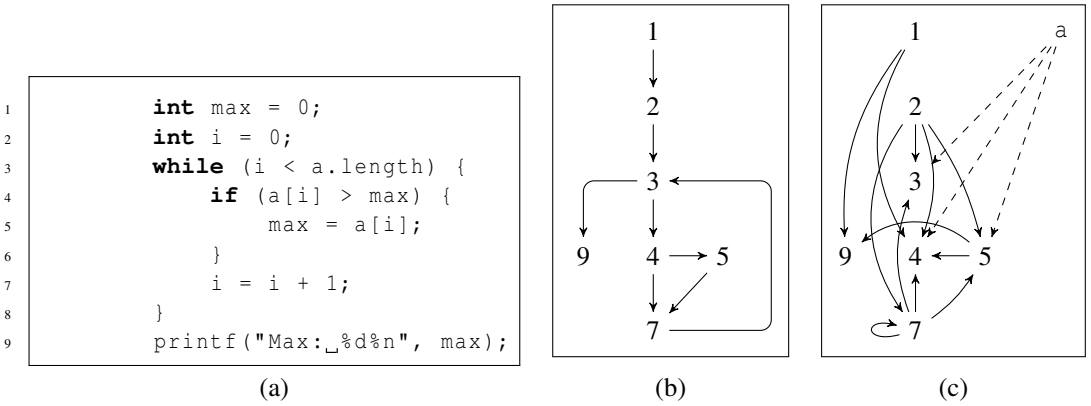

Figure 4.1: Control flow (b) and dependency graph (c) of a JAVA fragment (a)

graph, it can be seen that only 1 and 2 may be combined into a basic block, as all other nodes either have more than one incoming edge or more than one outgoing edge. From the data flow graph, it can be seen that within that block, statements 1 and 2 may be reordered, as there is no dependency between them.

Exercise 4-CC.2 Consider the following code fragment:

```
1  int x = in();
2  int i = 0;
3  boolean found = false;
4  while (!found && i < a.length) {
5      found = (a[i] == x);
6      if (!found) {
7          i = i + 1;
8      }
9  }
10 printf("Index: %d%n", i);
```

1. Draw the control flow graph and the data dependency graph.
2. According to the control flow graph, what are the basic blocks of this fragment?
3. According to the data dependency graph, which statements can be reordered? □

Exercise 4-CC.3 Consider the following code fragment:

```
1  int up = in();
2  int sum = 0;
3  for (int i = 0;
4           i < up;
5           i = i + 1) {
6      if ((i & 1) == 0) {
7          continue;
8      }
9      sum = sum + i;
10 }
11 printf("Sum: %d%n", sum);
```

If you have never seen the **`continue`**-statement, look it up!

1. Draw the control flow graph and the data dependency graph.
2. According to the control flow graph, what are the basic blocks of this fragment?
3. According to the data dependency graph, can any statements be reordered? □

In the next exercise you will write a tree visitor to build control flow graphs. Some code has been provided in the package `pp.block4.cc.cfg`:

- `Graph.java`: a simple control flow graph

- `Node.java`: node of a control flow graph, with fields for ID and number.
- `Fragment.g4`: Grammar for a fragment of JAVA that will compile the code snippets above.
- `BottomUpCFGBuilder.java`: Unfinished template for Exercise 4-CC.4
- `TopDownCFGBuilder.java`: Unfinished template for Exercise 4-CC.4

Control flow graphs can be constructed through top-down, inheritance-based rules or bottom-up, synthesis-based rules. In either case, the idea is that every parse subtree adds control flow nodes and edges to a global graph (given as an instance variable), but there is a `ParseTreeProperty` associating the correct entry and exit nodes.

Bottom-up: This is the most straightforward way. Every `exit`-method of the listener can build a control flow graph, based on the CFGs of its sub-statements. The construction is very much like that of an NFA from regular expressions. For simplicity, make sure that you stick to the convention given in the book that every CFG has a single entry point and a single exit point; then you only have to store the entry and exit points as attributes in your listener. The choice of having a single exit point will sometimes necessitate the creation of "fake" control flow nodes that do not correspond to any statement but just serve as exit nodes of some sub-CFG. Thus, the graphs constructed in this way will not be identical to the hand-drawn ones of the previous exercises.

Top-down: This is more tricky. Every statement tells all its children the entry and exit nodes they should use. For this purpose, the `enter`-methods of the listener should prepare those nodes and set them as attributes for their children. The `exit`-methods are unused.

Exercise 4-CC.4 Now program both solutions. *You may omit the* ***`break`*** *and* ***`continue`*** *statements; they are addressed in a challenge exercise below.*

1. Program the bottom-up CFG builder (template provided in `BottomUpCFGBuilder`)
2. Program the top-down CFG builder (template provided in `TopDownCFGBuilder`)
3. For both builders, show that they return correct flow graphs for the code snippets of Figure 4.1 above and Exercise 4-CC.2 (note that you do not have to declare array `a`). It is expected that the graphs are not *identical* to your answers to those exercises. What are the differences? □

Challenge JAVA knows the concept of *abrupt termination*: this is what happens if you **`break`** out of a loop or **`continue`** within that loop, and also if you **`return`** anywhere except at the end of a method body or when an exception gets **`throw`**n. All such phenomena disrupt the normal flow of control (and most are actually frowned upon by code purists for that reason); to get the CFG correct for such cases takes special care.

Exercise 4-CC.5 *This exercise is meant as a challenge and does not have to be signed off.*

Extend your bottom-up or top-town CFG builder (or both) from Exercise 4-CC.4 with methods for **`break`** and **`continue`** statements (both of which are already in the provided grammar `Fragment.g4`). For this purpose, you should realise that in fact every type of abrupt termination essentially requires its own exit node; those exit nodes can be synthesized or inherited in essentially the same way as the "regular" exits. Test out your program on the the JAVA snippets of Exercise 4-CC.3 and on the following (which computes the same value as the snippet of Exercise 4-CC.2):

```
int x = in();
int i;
for (i = 0; i < a.length; i++) {
    if (a[i] == x) {
        break;
    }
}
printf("Index: %d%n", i);
```

□

4-CC.2 ILOC

Appendix A of EC contains a specification of the linear intermediate representation format ILOC used in the book. Examples of the use of ILOC are scattered throughout the book. For instance, Fig. 5.8(b) shows how an array element is assigned in ILOC, and Fig. 5.14 shows more complicated ILOC code fragment. The JAVA snippet in Figure 4.1 is translated to the following ILOC code[1], in which `@a` denotes the offset of the beginning of array `a` from the address pointed to by `r_arp`, `@alength` denotes the length of the array `a`, and all other variables are kept in registers.

```
1  start: loadI 0 => r_max           // Line 1: max = 0;
2         loadI 0 => r_i             // Line 2: int i = 0;
3         loadI @alength => r_len
4  while: cmp_LT r_i,r_len => r_cmp  // Line 3: while (i < a.length)
5         cbr r_cmp -> body, end
6  body:  i2i r_i => r_a             // compute address of a[i]
7         multI r_a,4 => r_a         // multiply by size of int
8         addI r_a,@a => r_a         // add a's base offset
9         loadAO r_arp,r_a => r_ai   // r_ai <- a[i]
10        cmp_GT r_ai,r_max => r_cmp // Line 4: if (a[i] > max)
11        cbr r_cmp -> then,endif
12 then:  i2i r_ai => r_max          // Line 5: max = a[i];
13 endif: addI r_i,1 => r_i          // Line 7: i = i + 1;
14        jumpI -> while
15 end:   out "Max: ", r_max         // Line 9: out; not "official ILOC"
```

In the lab files you will find an assembler, a disassembler and a virtual machine for ILOC:

- `pp.iloc.Assembler` is an assembler. It has a singleton instance that; the method `parse` reads in ILOC programs in textual form (given as a string or in a file), such as the one shown above, and returns an object of type `pp.iloc.model.Program`.
- The method `prettyprint` of the class `Program` returns a string representation of the program, in the original syntax of ILOC.
- `pp.iloc.Simulator` simulates a `Program` by running it on a virtual machine (`pp.iloc.eval.Machine`) consisting of a simulated block of memory, a register map, and values for the symbolic constants `@a` etc.
- `pp.iloc.test` contains examples of the use of `Assembler` and `Simulator`.

Exercise 4-CC.6 Try out the assembler, disassembler and simulator on `max.iloc` by writing a test class.

1. Run the assembler on `max.iloc` and `prettyprint` the resulting `Program` object. If you parse the prettyprinted program again, the resulting `Program` should equal the one you got by parsing the original `max.iloc`.
2. Run the resulting `Program` in the `Simulator`. The method `Simulator#run` starts off the simulation; however, before calling it you should initialize the VM (which you can access through `Simulator.getVM`) by initializing the array `a`, i.e., giving values to the constants `@a` and `@alength` using `Machine#init` and `Machine#setNum`, respectively.
   Check that you get the expected output by programmatically inspecting the value of the register `r_max` after the simulation has finished, using `Machine#getReg`.
3. Explain the multiplication by 4 in Line 7 of the program. □

In the ILOC program above, the *basic blocks* correspond to the fragments starting with a label.

Exercise 4-CC.7 Draw the CFG of the ILOC program above, using the basic blocks as nodes, and compare it to the one in Figure 4.1. □

Exercise 4-CC.8 Answer Review Question 1 of Section 5.4 of EC. Ignore the fact that `fib` is a function: just implement the function body.

1. Write an ILOC program implementing `fib` based on a register-to-register model.

[1] Available in the lab files of Block 4 as `pp/block4/cc/iloc/max.iloc`

2. Write an ILOC program implementing `fib` based on a memory-to-memory model.
3. Write a test that shows that both programs return the same value for a range of input values of your choice. (For this purpose, you have to appropriately initialize the VM and inspect the result afterwards.)
4. What is the largest value for `n` for which the programs do not produce an integer overflow? □

Exercise 4-CC.9 Write ILOC code for the JAVA fragment in Exercise 4-CC.5. (It does not matter if you did not do that exercise.) To mimic the `in()`-call in the JAVA program, use the pseudo-ILOC-instruction **`in`**; see `pp.iloc.OpCode.in`. (Like the **`out`**-instruction demonstrated in the ILOC code fragment, this instruction is not part of the "official" ILOC: it is only provided for debugging purposes.)

Make sure your solution passes the provided `pp.block4.cc.test.FindTest` (which assumes that your code can be found in `pp/block4/cc/iloc/find.iloc`; change that to the correct filename). □

Finally, in the following exercise you will do some actual code generation (of ILOC code). This will be the first real compiler you build! Use the attribute rules in Fig. 4.15 (Page 207) of EC as inspiration.

Exercise 4-CC.10 Extend `pp.block4.iloc.CalcCompiler`, a tree listener for parse trees produced by the grammar `pp/block4/cc/iloc/Calc.g4` (which is actually identical to `pp/block3/cc/antlr/Calc.g4` of the previous Block). `CalcCompiler#compile` should generate ILOC code (in the form of a `Program` object) for an arbitrary `Calc`-expression, using a register-to-register memory model, ending with an `out` operation to print the calculated value. For instance, for the expression $1 + -3 * 4$, the generated code should be something like

```
loadI   1                => r_0
loadI   3                => r_1
loadI   4                => r_2
mult    r_1,r_2          => r_3
rsubI   0,r_3            => r_4
add     r_0,r_4          => r_5
out     "Outcome: ",r_5
```

Make sure your solution passes the `pp.block4.cc.test.CalcCompilerTest`. □