Block 5

Block 5

5-OV Overview

5-OV.1 Contents of This Block

CP This block will discuss concurrency models for homogeneous threading, where each thread executes the same instructions. We look at two programming techniques to support this: via compiler directives about possible parallellisations of a sequential program; and using dedicated hardware using graphics processing units (GPUs).

FP There is no new material in this block.

CC This block will discuss issues related to procedure calls and memory layout: how should local variables, global variables and parameters be stored and accessed so as to achieve the well-known call mechanism?

LP This block consists mainly of an exercise to let to you take full advantage of Prolog's unification and resolution to solve logical puzzles. The goal is to solve the "Ice Cream Tour" as elegantly as possible, i.e. by just specifying the problem and let Prolog come up with the solution, without exploring all possible solutions in a brute-force manner.

5-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;
- 2 hours self-study to read through the CC material and complete the lab exercises.

5-OV.3 Materials for this Block

- **CP**, The material for this week consists of several papers and book sections:
  - Barbara Chapman, Gabriele Jost and Ruud van der Pas. Using OpenMP – The Book. In any case, read Sections 4.3 (except 4.3.4), 4.4, 4.5 (except 4.5.3 and 4.5.4), 4.6.1, 4.6.3, 4.8.2, and 4.8.4. You can ignore all the bits about Fortran syntax. If you want to know more, Chapters 1 - 4 should give you the full basis.

  - The text on GPU programming in this manual (Section **??**).
  - Matthew Scarpino. *A Gentle Introduction to OpenCL*. Dr.Dobb's, August 2011.
  - For more information on OpenCL, see `http://streamcomputing.eu/knowledge/for-developers/tutorials/`.

  Links to the papers are available on Blackboard.
- **CC**, from EC: Chapter 6.

5-CP Concurrent Programming

The exercises for the CP strand will be added later.

5-FP Functional Programming

There is no new material for Block 5.

5-CC Compiler Construction

5-CC.1 ANTLR tree visitors

So far, we have used ANTLR to define *tree listeners*. You have seen that a tree listener combines a pre-order and a post-order traversal of the parse tree. In some cases, however, this is not powerful enough: it can happen that one needs an in-order traversal (where the parent node is visited after the first child or in between every pair of children) or even another, more dedicated traversal strategy. For this purpose, ANTLR also supports *tree visitors*. Tree visitors require a bit more work from the programmer but offer more control: essentially, they allow you to completely define your own traversal strategy.

The lab files for this block include a grammar `Building.g4` to illustrate the principle of tree visitors. To generate a tree visitor for this grammar, invoke ANTLR on `Building.g4` with the command-line arguments `-no-listener -visitor`. (See BLACKBOARD for directions on how to do this.) This will generate classes `BuildingVisitor` and `BuildingBaseVisitor` rather than the `BuildingListener` and `BuildingBaseListener` we have been working with so far.

In contrast to `BuildingListener`, the tree visitor interface `BuildingVisitor` provides not two but a *single* abstract method for every nonterminal: the dedicated method for a nonterminal `x` is called `visitX`. As parameter, each such method gets the same sort of context object as the `enter`- and `exit`-methods of a listener. Moreover, `BuildingVisitor` has a generic type; this is the return type of every `visitX`-method. To implement a visitor, extend `BuildingBaseVisitor` and override its `visitX`-methods, taking care of the following:

- Each `visitX` should call the global method `visit` on all children that need to be visited, in the order they need to be visited.
- In between the calls to `visit`, each `visitX` can do whatever it likes.
- Each `visitX` method should return a value of the instantiated generic type.

Exercise 5-CC.1 *This exercise does not need to be signed off; it serves as a preparation for the next ones.*
Study the provided `Building.g4` and `Elevator.java`

1. Generate the `Building` visitor classes using ANTLR.
2. Invoke `Elevator.java` on input of your choice. Study the `visitBuilding` and `visitFloor` methods and make sure you understand what they do. Note that `visitRoom` is never called (and hence it is not overridden in this class). □

Next, you'll program your own tree visitor. Consider the situation where an input string consists of an alternating sequence of numbers and words, such as for instance the text

```
4 strands 10 blocks 11 weeks 15 credits
```

and we want to print this on the standard output in the more readable form

```
4 strands, 10 blocks, 11 weeks and 15 credits
```

while simultaneously calculating the sum of the numbers occurring in the sentence (in this case, the sum would be 40). In the lab files, you will find a grammar `NumWord.g4` that can parse the input: the parser rules are

```
sentence: (number word)+ EOF;
number: NUMBER;
word: WORD;
```

(where `WORD` and `NUMBER` are token types with the expected definition).

Neither a preorder traversal nor a postorder traversal provides a natural way to print the required separators ", " and " `and` " between the groups of `number word`-pairs, so using a tree listener on this grammar it is not straightforward to program the desired behaviour. (It is not *impossible*: the exit method for `word` may inspect its parent and depending on whether that is a `sentence` and on the position of itself within that sentence print the correct separator.)

Exercise 5-CC.2 Implement the tree visitor `NumWordProcessor.java` (a skeleton of which is provided in the lab files) by overriding the `visitSentence`, `visitNumber` and `visitWord` methods so that the class has the required behaviour, described above (print the converted sentence, and return the sum of the numbers). You can test your `NumWordProcessor` by invoking the (provided) `main` method. □

Optional exercise: Faking the tree visitor. Alternatively, the functionality of a visitor can be faked through a listener by adding auxiliary intermediate nonterminals to the grammar, which are traversed by the listener at the right moments. For instance, the following rules, provided in the lab files in the form of the grammar `NumWordGroup`, generate the same language as `NumWord` above:

```
sentence
    : (group* penultimateGroup)? lastGroup EOF;

group
    : number word;

penultimateGroup
    : number word;

lastGroup
    : number word;

number: NUMBER;
word: WORD;
```

Here, the nonterminals `group`, `penultimateGroup` and `lastGroup` (which actually have exactly the same right hand sides) are defined with the sole purpose of allowing an "ordinary" tree listener to distinguish the right places in the sentence for the different separators.

Exercise 5-CC.3 *(This is an optional exercise and does not have to be signed off.)* Implement the tree listener `NumWordGroupProcessor.java` (a skeleton of which is provided in the lab files) by overriding the appropriate `enter-` or `exit-` methods so that the class has the required behaviour. Again, you can test your `NumWordGroupProcessor` by invoking the (provided) `main` method. □

Exercise 5-CC.4 *This is a follow-up to the optional Exercise 5-CC.3. It is meant for you to form your own opinion about the different solutions in Exercises 5-CC.2 and 5-CC.3; it does not have to be signed off.*

1. What advantages can you see of the visitor-based solution over the listener-based solution?
2. What advantages can you see of the listener-based solution over the visitor-based solution? □

Generating ILOC for control structures. We'll now combine a lot of the ingredients from Block 4 to create an ILOC code generator for a language that starts to look like you could actually do something with it. The provided grammar `SimplePascal.g4` defines a non-trivial fragment of the programming language `Pascal`, with the following features:

- The language supports integer and boolean types. On the ILOC level, booleans may be encoded as integers.
- All variables are defined up front, in a single, global scope. Upon declaration, variables are initialised to `0`, respectively `false` for booleans. Variables should be stored in memory; for this purpose, you have to calculate the offset of every variable with respect to the base of the scope. Remember that an integer uses four bytes.
- Besides assignments and blocks, the language supports **`If`**- and **`While`**-statements. This means that you have to generate conditional and immediate jumps (`cbr` and `jumpI`, respectively. For this purpose, you will need the tree visitor functionaltity introduced in Exercises 5-CC.1 and 5-CC.2.

Two sample PASCAL programs are provided together with the lab files:

- `gcd.pascal`, containing Euclid's algorithm for the computation of the greatest common divisor of two numbers;
- `prime.pascal`, containing a naive primehood detection algorithm.

Exercise 5-CC.5 Write a Simple Pascal program `gauss.pascal` that calculates the sum of all integers up to some upper bound set through an **`In`**-statement. *This question does not have to be signed off separately; you'll be asked to add this program to the tests of one of the next questions.* □

Because this language is no longer trivial, it becomes necessary to use a *two-pass compiler*. The first pass should achieve the following:

- It takes care of type checking;
- It calculates the offsets of the declared variables;
- It determines the entry points of the flow graphs corresponding to the statement and expression nodes of your parse tree.

The first pass (Exercise 5-CC.6) can be implemented using a tree listener. The results of this pass are collected into a single object that is used in the second pass (Exercise 5-CC.7 below) to generate ILOC code.

Exercise 5-CC.6 Complete the tree listener `pp.block5.cc.simple.Checker` so that it returns an instance of the class `Result` containing the outcome of the checker phase. Note that all expression-related listener methods have already been implemented; you only have to add the statement- and declaration-based methods. (Look into the grammar `SimplePascal.g4` to see which methods those are: as usual, everything that (transitively) occurs in the rules for `decl` and `stat` can potentially play a role in the calculation of the attributes.)

Test your solution using the provided `SimpleCheckerTest`. □

The second pass is more conventiently programmed as a tree visitor rather than a tree listener.

Exercise 5-CC.7 Complete the tree visitor `pp.block5.cc.simple.Generator` so that it generates (correct) ILOC code. Test your solution by running `SimpleCompiler` manually on the provided PASCAL files (`basic`, `fib`, `gcs`, `prime`) as well as your own solution to Exercise 5-CC.5, and also by running `SimpleGeneratorTest`.

To use the provided `Generator` optimally, consider the following:

- The generic type of the `TreeVisitor` has been instantiated to `Op`, meaning that every `visit` method returns a value of this type. You can take advantage of this to return the *first instruction* of the code emitted for a given parse tree node, which in turn can help you to find the correct (conditional and immediate) jump targets.
- To generate code for the conditional statements (**`If`** and **`While`**), you can apply the techniques you learned for the (bottom-up or top-down) control flow graph construction in Exercise 4-CC.4 of the previous Block.

□

5-CC.2 Dealing with procedures

Exercise 5-CC.8 Solve Exercise 6.3 from EC. □

Exercise 5-CC.9 In the PASCAL program of the previous exercise, consider the situation where

- `Main` has called `P1` (Line 33);
- `P1` has called `P4` (line 16);
- `P4` has called `P5` (line 30);
- `P5` has called `P1` (line 25).

Draw a graph showing the activation records at this point (including the one of `Main`), along the lines of Fig. 6.4 and 6.8, for each activation record including:

- The local data area (containing slots for each of the local variables — you do not have to invent values of the variables);
- The pointer to the caller's ARP;
- The access link (see Fig. 6.8). □

Exercise 5-CC.10 Solve Exercise 6.6 from EC, taking the following points into account:

- Figure 6.7 from EC gives an example of the kind of answer that is expected here.
- Show a single instance of `Elephant` and a single instance of `Dumbo`; choose your own values for the instance variables.
- The situation in JAVA differs from the one depicted in Fig. 6.7 in that the class of a class object is `Class`, whereas the superclass of every class that does not explicitly declare its own superclass is `Object`. Make sure you depict this situation correctly. □

Exercise 5-CC.11 Solve Exercise 6.8 from EC for the cases of call-by-value, call-by-reference and call-by-name (in other words, omitting call-by-value-result). For each of the simulations, show the content of the variables `a` and `i` after lines 6, 7, 8 and 9 of the program. You do not have to show the result of the print statements. □

Exercise 5-CC.12 Solve Exercise 6.10 from EC. Take into account that the ARs are stack-allocated here, and hence you have to actually combine them into a single stack. □

Exercise 5-CC.13 Solve Exercise 6.11 from EC. □

5-LP Logic Programming

This short week there will be only one practical exercise for Logic Programming, solving the Ice Cream tour. It is an instance of the (in)famous logic puzzles. The goal is to model this puzzle, and solve it by using the strength of Prolog's unification and backtracking. Note that backtracking over all possible solutions might be too slow, so the preference is to solve parts of the puzzle by just unification. We start with an "empty" solution and gradually fill it in by "matching" the partial solution with new facts. Sometimes, backtracking might be unavoidable.

5-LP.1 The Ice Cream Tour

The assignment is about solving a "logiquiz" on ice creams, which I found at `http://www.puzzles.com/projects/LogicProblems/IceCreamStands/Download.pdf`. The short description (taken verbatim from that website) is as follows:

> On her way home from work on Monday night, Sherry realized that on her daily commute she passed four ice cream stands. Each stand was in a different town along her route and the name of each was a different person's first name. Sherry loved ice cream and so felt compelled to try a different one on her way home each night for the rest of the week. Determine the name of

each ice cream stand, what town each stand was in, what day of the week she stopped at each, and what kind of ice cream cone she ordered at each one.

Here are the concrete clues:

1. Sally's Ice Cream wasn't in Rockland. Sherry didn't get peppermint stick ice cream on Thursday night.
2. Sherry stopped at the ice cream stand in Granite on the day before she got the chocolate chip ice cream and the day after she stopped at Gary's Ice Cream.
3. At Tom's Ice Cream she got peanut butter ice cream but not on Tuesday.
4. She got coffee bean ice cream on Wednesday, but not at Alice's Ice Cream.
5. She stopped at the stand in Marsh the day before she stopped at Sally's Ice Cream.

Actually, I can add one more piece of information. I called Sherry last night, and she told me that:

6. The ice she got in Boulder was gorgeous as well...

5-LP.2 Hints for this Assignment

The assignment is to let Prolog reconstruct Sherry's ice cream tour, based on the clues above. Use the power of unification, resolution and backtracking.

Question: Does this puzzle actually have a unique solution?

In principle, you are completely free how to model and solve this problem in Prolog. However, since the time for the Prolog project should be limited, some hints follow that might push you in some direction (hopefully saving time, but, unfortunately, also pruning some creativity). Feel free to diverge from this path!

Here are the hints:

1. You can model the problem as finding the ice-cream tour as an (ordered) list of the form:

```
[ stand(tuesday,_,...,_),
  stand(wednesday,_,...,_),
  stand(thursday,_,...,_),
  stand(friday,_,...,_) ]
```

   The list order is relevant to model the clues with "before" and "after".

2. Proceed as follows:

   (a) Figure out what fields are needed in the stand-structures in that list.

   (b) Write a predicate `icecream(Tour)`, constraining Tour by all clues.

   (c) Wrap your solution in a single **go(X)** predicate, such that the solution(s) to this logical puzzle will be generated in **X**.

3. Some last hints:

   (a) Write auxiliary functions on lists to express the before-after constraints conveniently;

   (b) Be careful with negative information: the negative statements contain positive information as well;

   (c) Remember that at the moment Prolog evaluates `not(P)`, the variables in `P` are supposed to be instantiated.

Wish you ice-cream weather and a lot of fun!