Block 1

Block 1

1-OV Overview

1-OV.1 Contents of This Block

FP This block will introduce some fundamental concepts and demonstrate some elementary examples of functional programs. Some of these concepts are: function, definition, currying, recursion, types, polymorphism.

CC This block will introduce the architecture of a compiler, and how to separate a piece of text into words and symbols (scanning). You will also meet ANTLR, the tool used in this course to automatically generate scanners and parsers.

1-OV.2 Mandatory Activities

The sign-off exercises of this block should be completed (and signed off) in the course of the block itself. If you are not finished, make sure that your solutions are ready to be signed off during the first lab session of block 2.

1-OV.3 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 2 hours self-study to read through the CC material and install ANTLR.

1-OV.4 Materials for this Block

- **CC**, from EC: Chapters 1 (completely) and 2 (except for 2.4.1–2.4.4 and 2.6).

1-FP Functional Programming

The material for FP is provided separately on BLACKBOARD.

1-CC Compiler Construction

1-CC.1 EC Chapter 1: Overview of Compilation

Exercise 1-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.

> *back end, front end, grammar, instruction scheduling, instruction selection, optimizer, parsing, register allocation, scanning, type checking*

□

Exercise 1-CC.2 Answer the following questions.

1. Using the grammar on Page 12 of EC, explain systematically, i.e., step by step, why the following string is an instance of the syntactic variable *Sentence*:
   "Most students is good programmers."
2. What is actually wrong with the above sentence? To what kind of programming error can you compare this? At what stage would a compiler detect it?
3. Which steps in the two sub-exercises above correspond to the following activities: *parsing*, *type checking*, *scanning*?

□

Exercise 1-CC.3 Extend the grammar on Page 12 of EC so that it can also cope with sentences of the form

> "Programming Paradigms is a diverse interesting module."
> "Programming Paradigms is a diverse interesting elective module."

without needing a new rule for every case where there are more successive adjectives. (A word like "a" is called a *particle* in English.) □

Exercise 1-CC.4 Consider the assignment

$$d \leftarrow d + 2 \times (a + b)$$

1. Manually carry out the (naive) instruction selection process informally described in §1.3.3 of the book (see Fig. 1.3) on this assignment.
2. Improve the sequence of instructions by minimizing the number of required registers
3. Improve your answer to the previous subquestion by rescheduling the sequence to a minimal execution time (where you may use more registers if that benefits the execution time). Compute the number of clock cycles of the original and the resulting schedule.

Give your answers in the form of tables such as the ones on Pages 17–19. □

1-CC.2 EC Chapter 2: Scanners

Exercise 1-CC.5 Answer Review Question 1 of Section 2.2 (Page 33), *with the proviso that the identifier should be exactly six characters*; i.e., for "zero to five alphanumeric characters" read "exactly five alphanumeric characters". □

Exercise 1-CC.6 Answer Review Question 2 of Section 2.3 (Page 42). Look for a concise RE; you may make use of all the notation introduced in the section. (The quotation mark, `"`, is an element of Σ.) □

Exercise 1-CC.7 Consider a language in with three different token kinds: (i) "`La`", (ii) "`La␣La`" and (iii) "`La␣La␣La␣Li`" (where the symbol ␣ denotes a space), with the proviso that

- Capitalisation must be as given: all `L`'s are upper case and all `a`'s and `i`'s lower case;
- The `a`'s may be repeated arbitrarily often (meaning there can be one or more), but the `i` may not; so `Laaaaaa` is allowed but not `Lii`;

- Every `La` and `Li` may be followed by zero or more spaces, *which are considered to be part of the token*.

Hence, for instance, "`LaaaaLaLaa␣Laaaa␣␣␣␣LaLiLaa`" is a valid input text; it consists of tokens "`LaaaaLa`", "`Laa␣Laaaa␣␣␣␣LaLi`" and "`Laa`".

1. Give regular expressions for the three token kinds of this language.
2. Give a single DFA that is able to recognise all three token kinds.
3. Answer the following questions, considering that scanning is *greedy*:
   - What is "`Laaaa LaLaa`" broken down into: "`Laaaa␣`" + "`LaLaa`", "`Laaaa␣La`" + "`Laa`" or "`Laaaa␣`" + "`La`" + "`Laa`"?
   - What is "`La␣La␣La␣La␣Li`" broken down into?

□

1-CC.3 Scanner implementation

Consider the following files, to be found on BLACKBOARD:

- `pp.block1.cc.dfa.State`. This is a straightforward implementation of a DFA — more precisely, a *state* of a DFA, but since all states are reachable from the initial state, it suffices to represent a DFA by its initial state. The class `State` includes a constant `DFA_ID6` that implements the 6-identifier scanner of Exercise 1-CC.5.
- `pp.block1.cc.dfa.Checker`. This is an interface offering the functionality of testing whether a given DFA accepts a given input text.
- `pp.block1.cc.test.CheckerTest`. This is a JUNIT test for an implementation of `Checker`.
- `pp.block1.cc.dfa.Scanner` This is an interface offering the functionality of applying a given DFA as a scanner to an input text.
- `pp.block1.cc.test.ScannerTest`. This is a JUNIT test for an implementation of `Scanner`.

Exercise 1-CC.8 Program an *efficient* implementation of `Checker`, and show it correct using `CheckerTest`. *Efficient* means that the execution time of your algorithm is *linear* in the length of the input string. You should be ready to argue why your solution is efficient. □

Exercise 1-CC.9 Program an *efficient* implementation of `Scanner`, and show it correct using `ScannerTest`. Efficiency here means the same as in Exercise 1-CC.8. Make sure that your solution correctly implements the notion of greediness, and that you are ready to argue why it is efficient. (*Hint:* reusing the `Checker` implementation of Exercise 1-CC.8 will *not* give rise to an efficient solution!) □

Exercise 1-CC.10 Add a constant `DFA_LALA` to `State`, analogous to `DFA_ID6`, that implements the DFA you developed in Exercise 1-CC.7. Add tests for this DFA to `CheckerTest` and `ScannerTest` that show the correctness of your DFA. □

1-CC.4 ANTLR

A grammar in ANTLR is defined in a file with extension `.g4`. In this course we will combine grammar files with JAVA source files. The ANTLR tool functionality consists of generating JAVA files from grammars that, when properly invoked, do the job of scanning, and later on, parsing and code generation.

This text assumes that you use ANTLR through its ECLIPSE plugin.

The lab files include the following example grammar:

```
1 lexer grammar Example;
2
3 @header{package pp.block1.cc.antlr;}
4
5 WHILE : 'while';              // Keyword
6 DO    : 'do';                 // Keyword
7 WS    : [ \t\r\n]+ -> skip ; // At least one whitespace char; don't make token
```

Here is an explanation of the most prominent features:

- Line 1: This declares the grammar. The name has to equal the file name (minus extension and not including any namespace information). The optional keyword **`lexer`** (which is used in the Compiler Construction world as a synonym for "scanner") means that this grammar only supports scanner rules; the default (obtained by leaving out the keyword) combines scanner and parser rules.
- Line 3: This specifies a line that should be inserted at the top of the Java files generated from the grammar. Since we want the generated files to be within the Java package "pp.block1.cc.antlr", we need a **`package`** declaration in the Java file; that is what this line achieves.
- Lines 5 and 6: This specifies that `while` and `do` are tokens of our `Example` language. `WHILE` and `DO` are the identifiers chosen for these tokens by the grammar designers; the fact that they equal the keywords is coincidental. Right now the names are themselves no used anywhere in the grammar.
- Line 7: This specifies that any non-empty sequence of characters from the set ' ' (space), '`\t`' (the TAB character), '`\r`' (the CR or Carriage Return character) and '`\n`' (the NL or Newline character) is considered a token; hoewever, the directive `->` **`skip`** then specifies that this token may actually be discarded. This construction is typically used for any information in the input file that does not need to be passed on for further processing: whitespace between other tokens (as here) or comments in the input file (in whatever comment syntax the input language supports).

Exercise 1-CC.11 To get acquainted with the ANTLR tool, carry out the following steps:

1. Generate the actual scanner `pp.block1.cc.antlr.Example` (a JAVA-file), and observe that, after this, all pre-defined lab files compile correctly. In the ECLIPSE plugin, this can be done by right-clicking the `g4`-file and selecting "Run As... → Generate Antlr Recognizer".
2. Study the syntax diagram of your language using the ECLIPSE plugin. For this purpose, go to "Window → Show View... → Other..." and select "ANTLR 4 → Syntax Diagram". This results in a so-called *railroad diagram* of your grammar rules; essentially a variation on a finite automaton where the labels are displayed as nodes and the nodes with incoming/outgoing edges as wiring between the nodes. Explain the difference between the railroad diagrams for `WHILE` and `WS`.
3. Run the JUNIT test `pp.block1.cc.test.ExampleTest`, and make sure you understand what's happening.
4. Run the JAVA class `pp.block1.cc.antlr.ExampleUsage`, and make sure you understand what's happening.

□

Exercise 1-CC.12 Consider again the grammar for 6-character identifiers from Exercise 1-CC.5.

1. Give an ANTLR lexer grammar that will recognize identifiers of this kind. Use so-called ANTLR *fragments* to make your rule(s) more readable (look up what fragments are in the online ANTLR documentation).
2. Test your grammar by providing a JUNIT test similar to `ExampleTest` (using the `LexerTester` class). Also test the acceptance of a sequence of identifiers. Does this behave as you expected?
3. What is the effect if you omit the keyword **`fragment`** from your grammar?

□

Exercise 1-CC.13 Consider again the grammar for PL/1 strings from Exercise 1-CC.6.

1. Give an ANTLR lexer grammar that will recognize strings of this kind, using the proper ANTLR syntax for exclusion (look it up!).
2. Test your grammar by providing a JUNIT test similar to `ExampleTest` (using the `LexerTester` class).

□

Exercise 1-CC.14 Consider the musical scanner of Exercise 1-CC.7.

1. Give an ANTLR lexer grammar that will recognize tokens of this kind.
2. Test your grammar by providing a JUNIT test.

□

1-CC.5 EC Section 3.3: Top-Down Parsing

Exercise 1-CC.15 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.

Sentential form, parse tree, ambiguity, left/right recursion, recursive-descent parsing, LL(x), bottom-up parsing, LR(x).

□

Exercise 1-CC.16 Regard the following variation on the grammar on Page 12 of the book:

1	*Sentence*	→	*Subject* `verb` *Object* `endmark`
2	*Subject*	→	`noun`
3	*Subject*	→	*Modifier Subject*
4	*Object*	→	`noun`
5	*Object*	→	*Modifier Object*
6	*Modifier*	→	`adjective`
7	*Modifier*	→	*Modifier Modifier*

After scanning, the sentence "all smart undergraduate students love compilers." gives rise to the token sequence

```
adjective adjective adjective noun verb noun endmark
```

Answer the following questions, where you may use the abbreviations:

S	*Sentence*	`n`	`noun`
U	*Subject*	`v`	`verb`
O	*Object*	`a`	`adjective`
M	*Modifier*	`e`	`endmark`

1. Give all leftmost derivations and all rightmost derivations of the above sentence, as well as the parse trees they generate.
2. Which parse tree best reflects the grammatical structure of the sentence?
3. How can you change the grammar so that it becomes unambiguous, and the only parse tree of the above sentence is the one you consider to be the best one?

□

Exercise 1-CC.17 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 1-CC.18 Make Exercise 3.4 from EC. *L* is the start symbol of the grammar. As in Exercise 1-CC.17, 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. □