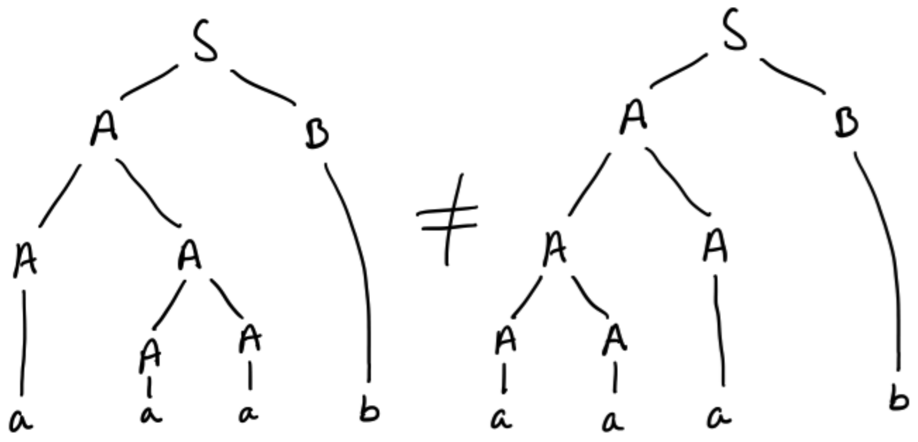
Individual Exercises - Lecture 5

1. Follow the studio associated with Crafting a Compiler Chapter 4
http://www.cs.wustl.edu/~cytron/cacweb/Chapters/4/studio.shtml
For this a virtual machine is provided on moodle with the necessary tools pre installed.

2. For each of the grammars below, describe the language associated with the grammar and determine if the grammar is ambiguous. If the grammar is ambiguous show two parse trees for the same string, otherwise explain why the grammar is not ambiguous (exercise 1 on page 168 in GE) – you may want to do this exercise by hand first and then try out the KfG tool in the AtoCC toolset: http://www.atocc.de/
   - a)
     - S -> A B
     - A -> A A | a
     - B -> B B | b

     Ambiguous. Example string: aaab

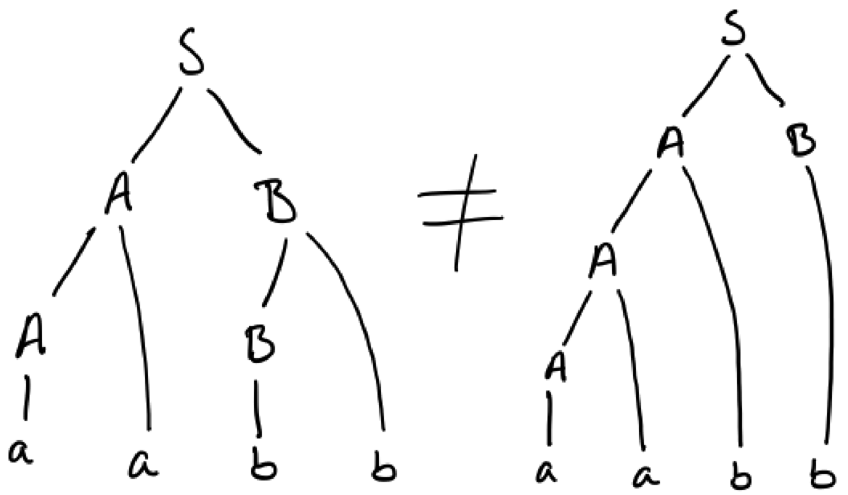


   - b)
     - S -> A B
     - A -> A a | a
     - B -> B b | b

     Not ambiguous

- c)
  - S -> A B
  - A -> A a | A b | a
  - B -> B a | B b | b

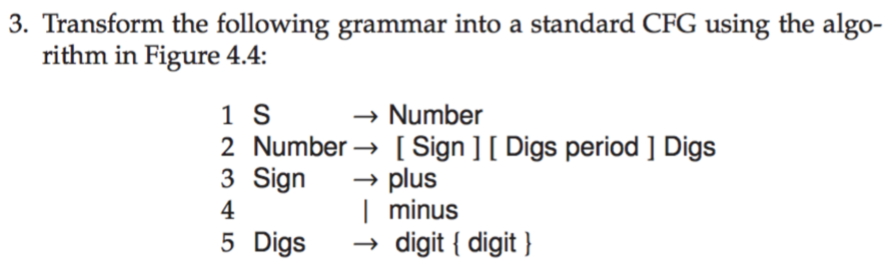
  Ambiguous. Example string: aabb



- d)
  - S -> A B
  - A -> A a
  - B -> B b

D: Not ambiguous, but can never reach sentential form anyway, since no derivation contains only terminals.

3. Do Fischer et al exercise 3 on page 138 and exercise 10 on page 5 (exercises 5 and 13 on pages 169-171 in GE)



3. Transform the following grammar into a standard CFG using the algorithm in Figure 4.4:

```
1  S       → Number
2  Number  → [ Sign ] [ Digs period ] Digs
3  Sign    → plus
4            | minus
5  Digs    → digit { digit }
```

The algorithm takes a CFG in EBNF and transforms it to BNF standard form.
First we expand Sign to be N1, which either derives Sign or nothing (lambda), since it was originally optional. Next we expand Digs periode to be Digs periode or nothing (lambda), to achieve optionality. Digs is expanded to understand repetition ({ }) by

adding a new non-terminal token, which can derive digits recursively.

Result:

1	S	$\rightarrow$	Number
2	Number	$\rightarrow$	N_1 N_2 Digs
3	N_1	$\rightarrow$	Sign
4		|	λ
5	N_2	$\rightarrow$	Digs period
6		|	λ
7	Sign	$\rightarrow$	plus
8		|	minus
9	Digs	$\rightarrow$	digit M
10	M	$\rightarrow$	digit M
11		|	λ

10. Compute First and Follow sets for each nonterminal in ac grammar from Chapter 2, reprised as follows.

1	Prog	$\rightarrow$	Dcls Stmts \$
2	Dcls	$\rightarrow$	Dcl Dcls
3		|	λ
4	Dcl	$\rightarrow$	floatdcl id
5		|	intdcl id
6	Stmts	$\rightarrow$	Stmt Stmts
7		|	λ
8	Stmt	$\rightarrow$	id assign Val ExprTail
9		|	print id
10	ExprTail	$\rightarrow$	plus Val ExprTail
11		|	minus Val ExprTail
12		|	λ
13	Val	$\rightarrow$	id
14		|	num

First(Prog) = {floatdcl, intdcl, id, print, $}

Follow(Prog) = {}

First(dcls) = {floatdcl, intdcl}

Follow(dcls) = {id, print, $}

First(dcl) = {floatdcl, intdcl)

Follow(dcl) = {floatdcl, intdcl, id, print, $}

First(stmts) = {id, print}
Follow(stmts) = {$}

First(stmt) = {id, print}
Follow(stmt) = {id, print, $}

First(ExprTail) = {plus, minus}
Follow(ExprTail) = {id, print, $}

First(Val) = {id, num}
Follow(Val) = {plus, minus, print, id, $}

4. (optional) Try the ACLA (Ambiguity Checking with Language Approximations) on the grammars from exercise 2 (exercise 1 on page 168 in GE). You can find the ACLA tool on http://services2.brics.dk/java/grammar/demo.html
   Did your answers agree with the answers from the tool?

5. (optional) You may also try Context Free grammar Tool on http://smlweb.cpsc.ucalgary.ca/start.html or its newest version on http://mdaines.github.io/grammophone/
   This is a good exercise for getting better at writing unambiguous grammars as well as getting a better understanding of when a grammar is LL(1), LR(0) etc.

6. (optional) Browse the Grammar Zoo web site http://slebok.github.io/zoo/
   Look up your favorite language

7. (optional) Browse the Syntax across languages web site: http://rigaux.org/language-study/syntax-across-languages/
   As an example, have a look at how many different ways a float can be declared!

8. (optional) Do Sebesta Review questions 1,2,4,5,8,10,24 on page 216-217

   1. What are the reasons why using BNF is advantageous over using an informal syntax description?
   It is very difficult (or even impossible?) to create an unambiguous description of a programming language using sentences in english. BNF allows us to specify grammar that can only be interpreted one way. A formal definition (fx in BNF) can help unify language definition across multiple compiler implementations..

   2. How does a lexical analyzer serve as the front end of a syntax analyzer?
   The lexical analyser performs analysis at the lowest level of program structure, i.e. the individual tokens / lexemes.

4. How can you construct a lexical analyzer with a state diagram?
The lexical analyzer can be constructed and modelled as a finite automata. The tokens of a programming language are a regular language, and a lexical analyser is a finite automaton.

5. Describe briefly the three approaches to building a lexical analyzer.
1. Write a formal description of the token patterns in a descriptive language related to regular expressions and then generate the lexical analyser from this. There exist many tools for this (the original being “Lex”).
2. Design a state transition diagram that describes the token patterns of the language and write a program that implements the diagram.
3. Design a state transition diagram that describes the token patterns of the language and hand-construct a table-driven implementation of the state diagram.

8. What are the two distinct goals of syntax analysis?
1. Check whether a given program is syntactically correct, and if not give proper error messages.
2. Produce a complete parse tree that can be used for translation.

10. Describe the recursive-descent parser.
A recursive-descent parser is a simple type of parser, that has a subprogram of each nonterminal in its associated grammar. Recursive-descent parsers do, however, have limitations, for example that it cannot handle left recursion in the grammar, due to the nature of recursive-descent parsers. They always expand from the left, which will obviously lead to stack overflow.

24. Why is a bottom-up parser often called a shift-reduce algorithm?
Bottom-up parsers are often called shift-reduce algorithms because shift and reduce are the two most common actions they specify.

Group Exercises - Lecture 5

1. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

2. Do Fischer et al exercise 1, 7, 8, 9, 13 on pages 138- 141 (exercises 4, 8, 10, 11, and 15 on pages 169-174 in GE)
   1. *While ambiguity is avoided in programming languages, (some) humor can be derived from ambiguity in natural languages. For each of the following English sentences, explain why it is ambiguous. First try to determine multiple grammar diagrams for the sentence. If only one such diagram exists, explain why the meaning of the words makes the sentence ambiguous.*

   (a) I saw an elephant in my pajamas.
   (b) I cannot recommend this student too highly.
   (c) I saw her duck.
   (d) Students avoid boring professors.
   (e) Milk drinkers turn to powder.

   (a) Was I wearing the pyjamas or was it the elephant?
   (b) Irony or not?
   (c) Is duck a verb or a noun?
   (d) Is boring an adjective or a verb?
   (e) Do milk drinkers turn INTO powder or turn to powder as an alternative?

7. *A grammar for infix expressions follows*

```
1  Start → E $
2  E     → T plus E
3        | T
4  T     → T times F
5        | F
6  F     → ( E )
7        | num
```

(a) Show the leftmost derivation of the following string.
num plus num times num plus num $

a)
E
T plus E
F plus E
num plus E
num plus T plus E
num plus T times F plus E
num plus F times F plus E
num plus num times F plus E
num plus num times num plus E
num plus num times num plus T
num plus num times num plus F
num plus num times num plus num

(b) Show the rightmost derivation of the following string.
num times num plus num times num $

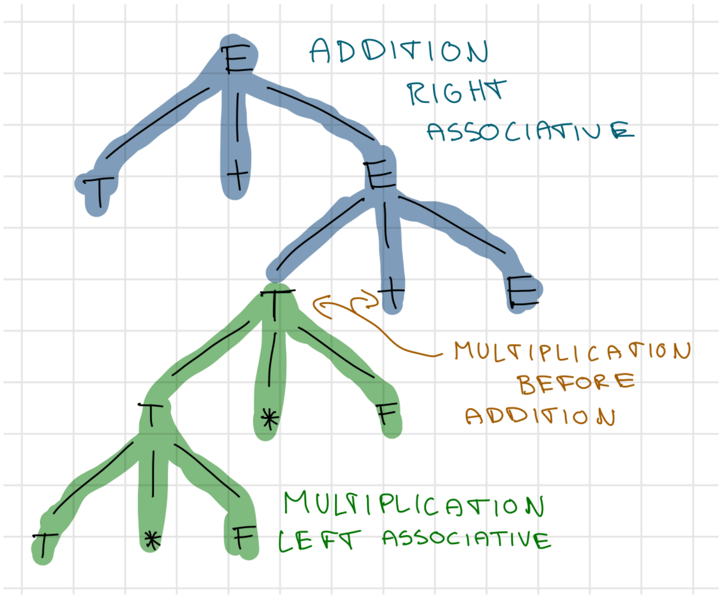
b) E
T plus E
T plus T
T plus T times F
T plus T times num
T plus F times num
T plus num times num
T times F plus num times num
T times num plus num times num
F times num plus num times num
num times num plus num times num

(c) Describe how this grammar structures expressions, in terms of the precedence and left- or right- associativity of operators.

Multiplication has precedence over addition (because it is located lower in the parse tree)

Addition is right associative (because addition expands right in the parse tree)

Multiplication is left associative (because multiplication expands left in the parse tree)


E
T
+
E
ADDITION RIGHT ASSOCIATIVE
T
+
E
MULTIPLICATION BEFORE ADDITION
T
*
F
T
*
F
MULTIPLICATION LEFT ASSOCIATIVE

8. Consider the following two grammars.

(a)

```
1  Start → E $
2  E     → ( E plus E
3        | num
```

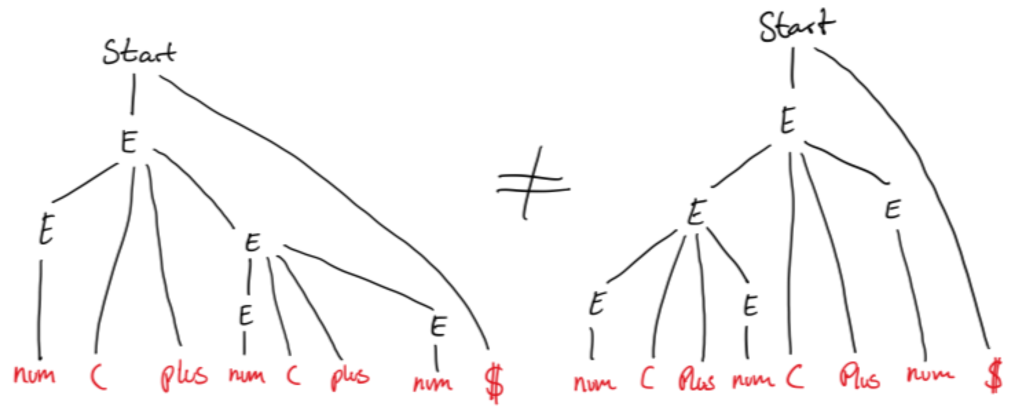
(b)

```
1  Start → E $
2  E     → E ( plus E
3        | num
```

Which of these grammars, if any, is ambiguous? Prove your answer by showing two distinct derivations of some input string for the ambiguous grammar(s).

(a) Not ambiguous

(b) It is ambiguous, proven by the following sentence “num (plus (num (plus num” which can be derived in two different ways yielding two different parse trees.



9. Compute First and Follow sets for the nonterminals of the following grammar.

```
1  S → a S e
2    | B
3  B → b B e
4    | C
5  C → c C e
6    | d
```

First(S) = {a, b, c, d}
Follow(S) = {e}

First(B) = {b, c, d}
Follow(B) = {e}

First(C) = {c, d}
Follow(C) = {e}