Individual Exercises - Lecture 7

1. (optional - recommended) Follow the studio associated with Crafting a Compiler Chapter 5
   http://www.cs.wustl.edu/~cytron/cacweb/Chapters/5/studio.shtml
   For this a virtual machine with the necessary tools pre installed is available on moodle.

2. Do Fischer et al exercise 1, 2 (d,e optional), 5, 6 (optional) on pages 173-178
   1. For each of the following grammars, determine whether the grammar is LL(1)

A grammar G is LL(1) iff
for each set of productions X ::= $X_1 \mid X_2 \mid \ldots \mid X_n$:
1. $first[X_1], first[X_2], \ldots, first[X_n]$ are all pairwise disjoint
2. If X_i =>* ε then $first[X_j] \cap follow[X] = \emptyset$, for $1 \leq j \leq n. i \neq j$

If G is ε-free then 1 is sufficient

(a)

1	S → A B c
2	A → a
3	| λ
4	B → b
5	| λ

(c)

1	S → A B B A
2	A → a
3	| λ
4	B → b
5	| λ

(b)

1	S → A b
2	A → a
3	| B
4	| λ
5	B → b
6	| λ

(d)

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

The algorithm says that for each non-terminal the predict set for its productions must be disjoint, where the predict set is defined as:

$$\text{predict}(A \to \alpha) = \begin{cases} \text{first}(\alpha) & \text{if } \alpha \text{ does not derive } \lambda \\ \text{first}(\alpha) \cup \text{follow}(A) & \text{if } \alpha \to^* \lambda \end{cases}$$

a)
First we create the predict set for A.
Predict(A -> a) = {a}
Predict(A -> λ) = {b, c}
This rule is valid for an LL(1) parser, since both productions are disjoint.

Predict(B -> b) = {b}
Predict(B -> λ) = {c}
This rule is valid for an LL(1) parser, since both productions are disjoint.

b)
Predict(A -> a) = {a}
Predict(A -> B) = {b}
Predict(A -> λ) = {b}
Since Predict(A -> B) == Predict(A -> λ), i.e not disjoint, this rule is not valid for an LL(1) parser, since it cannot distinguish the two options.
The grammar is also ambiguous.

c)

Predict(A -> a) = {a}
Predict(A -> λ) = {a, b}
Since a ∈ Predict(A -> a) and a ∈ Predict(A -> λ) this rule, and therefore the entire grammar, is not valid for an LL(1) parser.
This grammar is also ambiguous.

d)

Predict(S -> a S e) = {a}
Predict(S -> B) = {b, c, d}
No conflicts

Predict(B -> b B e) = {b}
Predict(B -> C) = {c ,d}
No conflicts

Predict(C -> c C e) = {c}
Predict(C -> d) = {d}
No conflicts

2. Consider the following grammar, which is already suitable for LL(1) parsing:

1	Start	$\rightarrow$	Value \$
2	Value	$\rightarrow$	num
3		|	lparen Expr rparen
4	Expr	$\rightarrow$	plus Value Value
5		|	prod Values
6	Values	$\rightarrow$	Value Values
7		|	λ

(a) Construct first and follow sets for each nonterminal in the grammar
First(Value) = {num, lparen}
First(Expr) = {plus, prod}
First(Values) = {num, lparen}

Follow(Value) = {$, num, lparen, rparen}
Follow(Expr) = {rparen}
Follow(Values) = {rparen}

(b) Construct Predict sets for the grammar

Predict(Value -> num) = {num}
Predict(Value -> lapren Expr rparen) = {lparen}

Predict(Expr -> plus Value Value) = {plus}
Predict(Expr -> prod Values) = {prod}

Predict(Values -> Value Values) = {num, lparen}
Predict(Values -> λ) = {rparen}

C) Construct recursive-descent parser based on the grammar

Here is a pseudo code version of a recursive-descent parser. In the comments of the code you can see why we peek for the tokens, that we do. We use the mechanical approach of first checking the predict set for each production to tell which one should be used. It is done in the order in which they appear in the grammar. For lambda productions the follow-set is checked.

```
Procedure Start()
      call Value()
      call Match(ts, $)
end
```

```
Procedure Value()
      // Predict(Value -> num) = {num}
      if ts.peek() = num
      then
            call Match(ts, num)
      // Predict(Value -> lapren Expr rparen) = {lparen}
      else if ts.peek() = lparen
      then
            call Match(ts, lparen)
            call Expr()
            call Match(ts, rparen)
      else
            call ERROR()
end
```

```
Procedure Expr()
      // Predict(Expr -> plus Value value) = {plus}
      if ts.peek() = plus
      then
            call Match(ts, plus)
            Value()
            Value()
      // Predict(Expr -> prod Values) = {prod}
      else if ts.peek() = prod
      then
            call Match(ts, prod)
            call Values()
      else
            call ERROR()
end
```

```
Procedure Values()
      // Predict(Values -> Value Values) = {num, lparen}
      if ts.peek() ∈ {num, lparen}
      then
            call Value()
            call Values()
      else
            // Follow(Values) = {rparen}, due to lambda production
            if ts.peek() ∈ {rparen}
            then
                  /* Do nothing for lambda-production */
            else
                  call ERROR()
end
```

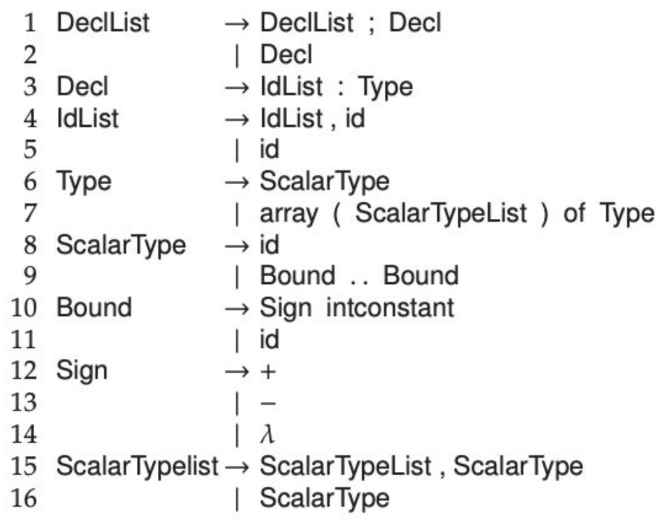
D) (OPTIONAL) Add code into the parser to compute sums and products as indicated

You could change the Value() and Expr() to return a number, and change Values() to return the list of numbers that are read.

Inside the Expr() function you could then add or multiply the numbers returned from Value()/Values() calls based on whether the token you read is a ‘plus’ or ‘prod’.

E) (OPTIONAL) Build an LL(1) parse table on the grammar
A parse table tells us which rule to use depending on which rule we are inside and which token we get as the next one. You can use the predict sets to construct this table.
There is very good example of an LL(1) parser using a parse table here:
http://www.cs.ecu.edu/karl/5220/spr16/Notes/Top-down/LL1.html

5. Transform the following grammar into LL(1) form using the techniques presented in Section 5.5:



```
 1  DeclList       → DeclList ; Decl
 2                 | Decl
 3  Decl           → IdList : Type
 4  IdList         → IdList , id
 5                 | id
 6  Type           → ScalarType
 7                 | array ( ScalarTypeList ) of Type
 8  ScalarType     → id
 9                 | Bound .. Bound
10  Bound          → Sign intconstant
11                 | id
12  Sign           → +
13                 | −
14                 | λ
15  ScalarTypelist → ScalarTypeList , ScalarType
16                 | ScalarType
```

Grammars are not LL(1) because of prediction conflicts. Two common causes of prediction conflicts are left recursion and common prefixes. This grammar contains both. Productions 1, 4 and 15 are left recursive; they will need to be rewritten. More subtly, productions 8 and 9 share a common prefix, id. This is because Bound can generate an initial id. Hence productions 8 and 9 will need revision.

Method EliminateLeftRecursion in Figure 5.15 on page 157 transforms left-recursive productions into an equivalent LL(1) form. The idea is that a left-recursive production may be used zero or more times before another non- left-recursive production is used. The net effect is to generate the result of the non-left-recursive production followed by zero or more occurrences of the left-recursive production's "tail." Thus for productions 1 and 2, a Decl must be produced by production 2 followed by zero or more "tails" of production 1 "; Decl". So production 1 is rewritten to first generate Decl followed by a new non-terminal, DeclListTail. DeclListTail generates zero or more occurrences of "; Decl". The rewritten productions are:

```
DeclList     → Decl DeclListTail
DeclListTail → ; Decl DeclListTail
             | λ
```

Similar transformations are applied to productions 4 and 5, and 15 and 16, yielding:

```
IdList      → id IdListTail
IdListTail →  , id IdListTail
               |  λ

ScalarTypeList       → ScalarType ScalarTypeListTail
ScalarTypeListTail → , ScalarType ScalarTypeListTail
                           |   λ
```

To solve the prediction conflict between productions 8 and 9, we split production 9 into two alternatives: a bound that begins with an identifier and a bound that begins with an integer. Then the two productions that begin with id are combined into one, with a BoundSuffix that generates the two valid bound suffixes. We have

```
ScalarType → id BoundSuffix
                |  Sign intconstant .. Bound
BoundSuffix → .. Bound
BoundSuffix → λ
```

6. (Optional) Run your solution to Exercise 5 through any LL(1) parser generator to verify that it is actually LL(1). How do you know that your solution generates the same language as the original grammar?
Try using this online checker:
https://www.cs.princeton.edu/courses/archive/spring20/cos320/LL1/

This will check if the grammar is LL(1) and also generate a transition table.

The question of whether two CFGs describe the same language is very difficult. In fact, this class of problems is undecidable.

3. (Optional - but recommended for hands-on experience) Familiarize yourself with JavaCC, Coco/R and ANTLR by browsing their web pages
The virtual machine provided on Moodle has these tools available, along with a set of guides that walks you through a small example language for each tool.

Group Exercises - Lecture 7

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

2. Do Fischer et al exercises 9 and 10 on pages 173-178 (exercises 9 and 12 on pages 205-210 in GE)

   9. Recall that an LL(k) grammar allows k tokens of lookahead. Construct an LL(2) parser for the following grammar:

```
1  Stmt   → id ;
2         | id ( IdList ) ;
3  IdList → id
4         | id , IdList
```

```
Procedure Stmt()
    if ts.peek(2) == [id, ';']
         call match(ts, id)
         call match(ts, ';')
         // end of program
    else if ts.peek(2) == [id, '(']
         call match(ts, id)
         call match(ts, '(')
         IdList()
         call match(')')
         call match(';')
    else
         call error()

Procedure IdList()
    if ts.peek(2) == [id, ',']
         call match(ts, id)
         call match(ts, ',')
         IdList()
    else
         call match(ts, id)
```

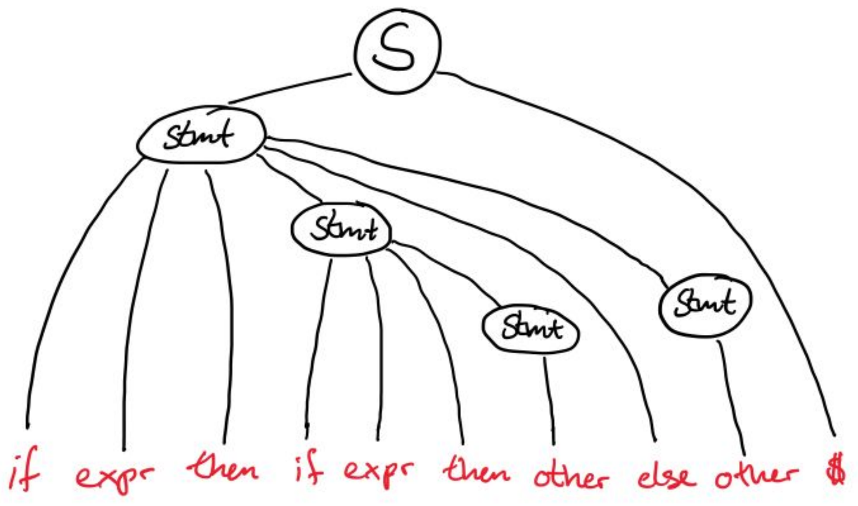
10. Show the two distinct parse trees that can be constructed for

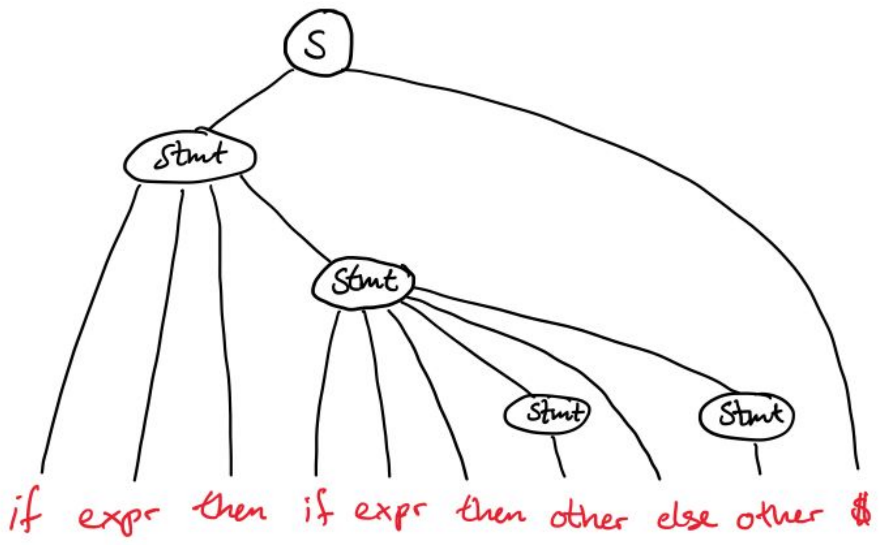
if expr then if expr then other else other

using the grammar given in Figure 5.17. For each parse tree, explain the correspondence of then and else.

```
1  S    → Stmt $
2  Stmt → if expr then Stmt else Stmt
3       | if expr then Stmt
4       | other
```

Figure 5.17: Grammar for if-then-else.





In the first tree the else clause is bound to the first if clause. In the second tree, the else clause is bound to the inner if clause.

3. Discuss pros. and cons. of writing a recursive descent parser by hand and using a tool
   Pros (of writing by hand):
   - You can add features, that would otherwise not be traditional feature of a parser
     - See: Python indentation tokens for scopes etc.
     - Or prettification of tokens to make them more readable
   - You get a better understanding of the inner workings of the parser

   Cons:
   - If the language is big with many constructs and tokens, it can take a long time and leaves plenty of room for error to write it by hand.
   - Constructing a parser is a very mechanical process, so if you understand the inner workings, there is no reason to repeatedly implement one.
   - Changes to the syntax of the language is more difficult with a hand crafted parser, whereas a parser generated would simply generate a new one from the modified grammar.

4. Discuss pros. and cons. of JavaCC, Coco/R and ANTLR
   Try out the tools inside the virtual machine provided on moodle! On the Desktop you will find pdf guides that walk you through a simple example for each tool.

   Other than the implementation experience itself, other pros include the fact that the generated parsers often include a lot of optimisations which make them very fast.

5. (optional - but recommended) Follow the Lab associated with Crafting a Compiler Chapter 5 http://www.cs.wustl.edu/~cytron/cacweb/Chapters/5/lab.shtml
   You can use the virtual machine provided on Moodle.