Universitetet i Oslo
Institutt for Informatikk
Reliable Systems
Martin Steffen

INF 5110: Compiler construction

Spring 2024 **Series 3** 26. 2. 2024

Topic: Chapter 4: grammars (Exercises with hints for solution)

Issued: 26. 2. 2024

Exercise 1 (LL(1)) Check if the following grammar is LL(1)?

$$S \;\rightarrow\; (\,S\,)\,S \;\mid\; \epsilon$$

Solution: To answer that question we need to calculate the *first* and the *follow* sets. One could notice that the grammar has an ϵ-production. That's relevant insofar in that it complicates slightly the check if the grammar is LL(1) or not; but in this case, the given grammar is rather simple, anyhow. It would be easier without nullable non-terminals and of course, an ϵ-production makes the corresponding non-terminal on the left-hand side immediately nullable (and in general, potentially further symbols, as well).

Unlike LR-parsing, LL-parsing, for instance in the form of recursive descent parsing, is actually pretty simple to understand and to remember, even with nullable symbols. This form of parsing does a *left-most* derivation and a top-down build-up of the parse tree. Generally, a grammar is LL(1), if that kind of derivation can be done deterministically. As mentioned in the lecture, for derivations in general, there are 2 kinds of sources of non-determinism at each point: 1) where to apply a production, and 2) which production to apply. Point 1) is irrelevant, as a left-to-right derivation simply chooses the left-most non-terminal. So we don't have to worry about that. Remains the "which-production-to-chose" non-determinism, more precisely: "which-right-hand-side-to-chose", because the non-terminal is fixed (it's the left-most …).

How can one make that choice, resp., under which circumstances is a unique choice possible? Assuming that the non-terminal in question is A, LL(1) top-down parsing relies on *one look-ahead* to make unique choices, and we have to check, given the grammar, that this is possible. Given a non-terminal A with some right-hand sides, the question is roughly if, given two different right-hand sides, will the ultimately resulting possible words have *disjoint* sets of terminals *to start with.* This is only a rough description, the lecture was more precise (for instance, in the precise definition, A is not considered in isolation, but as part of a left-most derivation. The technical term was *left sentential form*).

In absence of nullable symbols, that makes the check immediate. Actually it means that we can consider basically the A in isolation: the symbols that the ultimate word can start with are given by A's *first set*, which leads to the following check, based only on the first sets:[1]

[1]The property uses the fact that the grammar is "reduced". We did not cover that in a focused manner in the lecture. It basically means that the grammar does not contain "useless symbols", in particular useless non-terminals. Especially, if A would never appear in any derivation, as it's unconnected from the start symbol or no word of terminals could ever be derived from it, then the follow sets of A would be irrelevant and would have no influence on the question whether the grammar is LL(1) or not.

Lemma 1 (LL(1) (without nullable symbols)) A reduced context-free grammar without nullable non-terminals is an LL(1)-grammar iff for all non-terminals A and for all pairs of productions $A \to \alpha_1$ and $A \to \alpha_2$ with $\alpha_1 \neq \alpha_2$:

$$\mathit{First}_1(\alpha_1) \cap \mathit{First}_1(\alpha_2) = \emptyset \ .$$

In the presence of nullable symbols, things get slightly more tricky: the question what comes "first" after a right-hand side of A is *not only* determined by the things derivable from A (captured by the first-set), but also by what *follows* A. Thus, in the more complex, general setting, the check becomes as follows:

Lemma 2 (LL(1)) A reduced context-free grammar is an LL(1)-grammar iff for all non-terminals A and for all pairs of productions $A \to \alpha_1$ and $A \to \alpha_2$ with $\alpha_1 \neq \alpha_2$:

$$\mathit{First}_1(\alpha_1 \mathit{Follow}_1(A)) \cap \mathit{First}_1(\alpha_2 \mathit{Follow}_1(A)) = \emptyset \ .$$

Be carefull with the definition. It's

$$\mathit{First}(\alpha_1\ \mathit{Follow}(A)), \quad \textit{not} \quad \mathit{First}(\alpha_1)\ \mathit{Follow}(A) \ .$$

Without consulting any book and without remembering the lemma here, one way of solving the current task is: build the LL(1)-parsing table and see of it contains "double entries". For the corresponding recipe, see the slide *"Construction of the parsing table"* in the top-down parsing section. The **LL-table recipe** from the lecture, of course, directly embodies the principles of the lemma.

Assume $A \to \alpha \in P$.

1. If $\mathbf{a} \in \mathit{First}(\alpha)$, then add $A \to \alpha$ to $M[A, \mathbf{a}]$.
2. If α is *nullable* and $\mathbf{a} \in \mathit{Follow}(A)$, then add $A \to \alpha$ to $M[A, \mathbf{a}]$.

Armed with those reminders, the **battle plan** concretely for the task is:[2]

Start by calculating the first- and follow sets.

That gives the following table:

	First	*Follow*
S	ϵ, **(**	**$**, **)**

Afterwards, with that information, make a "decision table" for the non-terminal S. The following corresponds to the LL(1) parsing table (here very simple, with one line only, as we have only one non-terminal):

The table has **no duplicate entries:** for each combination of nonterminal (only S here) and terminal (the 1-look-ahead), there's not more than one entry. In more general cases, of course, some entries may be empty, which would correspond to an error situation, but that would not indicate ambiguity.

Let's comment on the table a bit. The very small table has no empty slots, so how does a parser reject input? Well, just because the table has empty slots does not mean all input is accepts. In the first column, the slot is $S \to \mathbf{(}\ S\ \mathbf{)}$. that means, that the parser expects and eats

[2]Don't forget the ϵ for the first set and the **$** for the follow set. As might be seen from the above lemmas, in absence of nullable symbols, one can get away with calucating the first-sets only.

	(	)	$
S	$S \to (\,S\,)\,S$	$S \to \epsilon$	$S \to \epsilon$

Table 1: LL(1) parsing table

(first. That is guaranteed, because that colums is chosen by the look-ahead of 1, namely in the case that (is the next token. After successfully parsing S, it expects the matching) afterwards, and if that is not the case, an error will be raised.

What about) as input, which certainly is syntactically wrong? Well, we have to look more careful at how the LL(1) parser works, in particular how it rejects input. Initially, the stack contains S (besides **$**). The next input is), the stack content S is replaced by ϵ. That means the stack contains only **$**. Since the input in not yet finished as well, the parser reports an error.

Here's how one could write in some form of pseudo-code a recursive descent "parser" (without AST generation etc.) for that grammar:

Listing 1: recursive descent procedure

```
procedure S() {
    if token = "("
    then getToken();
          S()
          match (")");
          S()
    else skip
    end
end
```

Similar to the remarks s in connection with the parsing table, we should keep in mind how a parser accepts or rejects thing. In the same way that the parsing table does not give the full picture, also the procedure from Listing 1 is not all of the parser. Using the same example again: using) as input, the procedure simply terminates, but since the input is not yet empty, the parser reports an error.

As some extra bit of info for the interested (it's not part of the question): let's look at the following grammar:

$$S \;\to\; S\,(\,S\,) \mid \epsilon$$

It's pretty obvious that this reformulation of the original grammar does not change the *language*. Going through the same motions as before and doing the first and follow sets gives

	First	*Follow*
S	ϵ, (	**$**, (,)

Well, that means this *grammar* is *not* LL(1). One can find that out by doing the analogous table-construction; ultimately, the problem now is the "overlap" on the (-symbol.

The *language* still is LL(1), of course, since there *exists* a grammar for the language which is LL(1), namely the first one. One difference in the grammars is that the latter is left-recursive, whereas the first one is not. Actually, if a grammar is left-recursive, it can't be LL(1) or LL(k), so even without explicitly calculating the first- and follow-sets, the answer could have been clear. □

Exercise 2 (Ambiguity) Given the following grammar.

$$\begin{array}{lcl} exp & \rightarrow & exp + exp \mid (\, exp\,) \mid \textbf{if}\ exp\ \textbf{then}\ exp\ \textbf{else}\ exp \mid var \\ var & \rightarrow & \ldots \end{array}$$

1. Try to come up with an *unambiguous* grammar for the language of the given grammar, where
   (a) addition is left-associative, and where
   (b) **if** x **then** y **else** $z + y$ is meant to mean **if** x **then** y **else** $(z + y)$.
2. Why don't we have a dangling else problem here?

Solution:

1. We should also note as an aside: we have seen grammars for conditionals in the lecture, but they are for conditionals as **statements**. In the exercise here, we are dealing with *conditional expressions* which turn out to be at the root of the problem.

   We could start by identifying a plausible cause of the ambiguity problem, and it's clear that addition is ambiguous. I's supposed to be left-associative, so we could add a new non-terminal, say exp', and reformulate the grammar as follows:

$$\begin{array}{rcl} exp & \rightarrow & exp + exp' \mid exp' \\ exp' & \rightarrow & (\, exp\,) \mid \textbf{if}\ exp\ \textbf{then}\ exp\ \textbf{else}\, exp \mid var \\ var & \rightarrow & \ldots \end{array}$$

   This was a solution proposed in some earlier semesters. On closer inspection, though, it turns out that this grammar is still ambiguous, i.e., addition is not the only reason for ambiguity! As an example, take the following expression

$$\textbf{if a then b else c} + \textbf{d} \ .$$

   With only one addition operation, there is no issue whether $+$ associates to the left or right. The (remaining) problem is the interplay between addition and the conditional and the the token stream may be interpreted as

$$\textbf{if a then b else } (\textbf{c} + \textbf{d}) \quad \text{or as} \quad (\textbf{if a then b else c}) + \textbf{d} \ ,$$

   where the parentheses are used to indicate the parse tree. That means that, in a way, there is a "dangling-plus" problem. Unlike the dangling-else problem, that's not standard terminology. The general situation, however, that an "if-then-else"-construct can be part of expressions, not just of statements (where expression then include more than just "plus" of course) is not uncommon.

   Maybe this, but the other way around

$$\begin{array}{rcl} exp & \rightarrow & \textbf{if}\ exp\ \textbf{then}\ exp\ \textbf{else}\, exp \mid exp' \\ exp' & \rightarrow & exp' + exp'' \mid exp'' \\ exp'' & \rightarrow & (\, exp\,) \mid var \end{array}$$

$$\begin{array}{rcl} exp & \rightarrow & \textbf{if}\ exp\ \textbf{then}\ exp\ \textbf{else}\, exp' \mid exp' \\ exp' & \rightarrow & exp' + exp'' \mid exp'' \\ exp'' & \rightarrow & (\, exp\,) \mid var \end{array}$$

$$\begin{array}{rcl} exp & \rightarrow & exp + exp' \mid exp' \\ exp' & \rightarrow & \textbf{if}\ exp\ \textbf{then}\ exp\ \textbf{else}\, exp' \mid exp'' \\ exp'' & \rightarrow & (\, exp\,) \mid var \end{array}$$

2. That's easy: unlike the grammars in the lecture, the **else** part is not optional, but mandatory. As a consequence, there is no dangling-else problem. Nonetheless, there's a "dangling-plus" problem, as described.

□

Exercise 3 (Ambiguity) Given the following grammar.

$$\begin{array}{rcl} exp & \to & exp\ op\ exp \mid (\ exp\ ) \mid \mathbf{number} \\ op & \to & + \mid - \mid * \mid / \mid \uparrow \mid < \mid = \end{array}$$

Do the following things.[3]

1. The grammar is pretty ambiguous. Make an unambiguous grammar capturing the same language, under the following side conditions

	precedence	assoc
$\uparrow$	highest (3)	right
$*$, $/$	level 2	left
$+$, $-$	level 1	left
$<$, $=$	0	non-associative

2. Give recursive-descent procedures for each non-terminal to check the grammar (using also loops, if advisable). Divide the terminals representing *op* in an appropriate manner
3. Based on the previous point: add tree-building code into the procedures in such a way that sequences of exponentiations $\uparrow$ are treated appropriately in the sense that the tree reflects the intended right-associativity.
4. Take the unambiguous grammar done in the first point, remove left-recursion, and do left-factorization (without destroying unambiguity).
5. Check whether the resulting grammar is LL(1).

Solution:

1. Introducing precedences (to deal with the ambiguity) means, there are expressions at different levels of precedence. One should also remember resp. compare the quite similar Exercise 3 from the previous Series 2. Compared to that one, the exercise *here* adds the **non-associative relations** at precedence level 0, the rest is unchanged. In the lecture, there were 2 precedence levels and 3 different levels of expressions (called expressions, terms, and factors), in the exercises from Series 2, there were 3 precedence levels and we had, in that earlier exercise, 4 different versions of expressions).

   Now we have 4 levels of precedences, still one more, which gives *5* different "kinds" of expressions.

   Note also: the question is kind of not well-formulated. It is not clear what we mean by non-associativeness, resp. how to deal with it. If it's dealt with in that the parser should interpret a $1 = 2 = 4$ as both $(1 = 2) = 3$ and $1 = (2 = 3)$, then the grammar will be ambiguous, despite the fact that the task also requires the grammar to be unambiguous. If we interpret "non-assoc" as "do as you like, left-assoc or alternatively right-assoc, both are fine". Then one can make an unambigous grammar.[4]

[3]There's a certain amount of repetition here, we won't go through everything during class-time, but a proposal for solution will be available.

[4]A third interpretation of non-associativity might be: the symbols are non-associative, therefore it's simply not allowed to write $1 = 2 = 3$, the user has to make it clear what's meant, using parentheses.

To reflect the precedences, the grammar needs to introduce new non-terminals to represent those levels. The original level *exp* still remains, which mean we end up with the following "versions" of expressions

exp	as originally given
exp_1	operands before or after $<$ and $=$
exp_2	operands before, between, or after $-$ and $+$ ("term")
exp_3	operands before, between or after $*$ and $/$ ("factor")
exp_4	operands before, between or after $\uparrow$

$$\begin{array}{rcl} exp & \rightarrow & exp\ \mathbf{relop}\ exp_1 \ \mid\ exp_1 \\ exp_1 & \rightarrow & exp_1\ \mathbf{addop}\ exp_2 \ \mid\ exp_2 \\ exp_2 & \rightarrow & exp_2\ \mathbf{mulop}\ exp_3 \ \mid\ exp_3 \\ exp_3 & \rightarrow & exp_4\ \mathbf{eop}\ exp_3 \ \mid\ exp_4 \\ exp_4 & \rightarrow & (\ expr\ ) \ \mid\ \mathbf{number} \end{array} \tag{1}$$

reformulate

The following is a *wrong* solution (depending on how one interprets the question), which had been around for some years (no one, including me the first year, noticed . . .)

$$\begin{array}{rcl} exp & \rightarrow & exp_1\ \mathbf{relop}\ exp_1 \ \mid\ exp_1 \\ exp_1 & \rightarrow & exp_1\ \mathbf{addop}\ exp_2 \ \mid\ exp_2 \\ exp_2 & \rightarrow & exp_2\ \mathbf{mulop}\ exp_3 \ \mid\ exp_3 \\ exp_3 & \rightarrow & exp_4\ \mathbf{eop}\ exp_3 \ \mid\ exp_4 \\ exp_4 & \rightarrow & (\ expr\ ) \ \mid\ \mathbf{number} \end{array} \tag{2}$$

What is wrong is that an expression

$$1 = 1 = 1 = 1$$

cannot be parsed (which was possible in the original grammar). It should be noted: to forbid such syntax is in-line with the requirements from the compila language from the *oblig*, where it's specified that such relational expressions are considered as syntactically disallowed. Perhaps one could make the argument: stating that relations are non-associative *implies* that $1 = 1 = 1 = 1$ is disallowed. In that interpretation, the latter grammar was not wrong either.

In the grammar the **relop** represents $<$ and $=$ but is meant as one token (i.e. terminal), were $<$ and $=$ are two different token values. Alternatively we might have used a non-terminal *relop* and more productions. Analogous remarks apply to the treatment of the other operators.

Besides the precedences: important is to understand how one gets the *associativity correct.*

2. This is a sketch of code for `exp3` as an example. In the code, the ascii representation of the exponent $\uparrow$ is "`**`"; Note: the `exp3` is the *right-associative* construct, unlike the others.

   The *lesson* here is the comparison with the treatment of the left-associative cases (see above). Those are kind of easy, they mesh well with the recursive calls. For the right-associative case, it's more tricky, it involves a bit fiddling with the trees. Furthermore, we make use of a *while-loop*. Anyway, here we go: A typical tree may look like this:

```
  e3
 /  \
e4   e3
|   /  \
n e4   e3
  |   /  \
  n  e4    e4
      |    |
      n    n
```

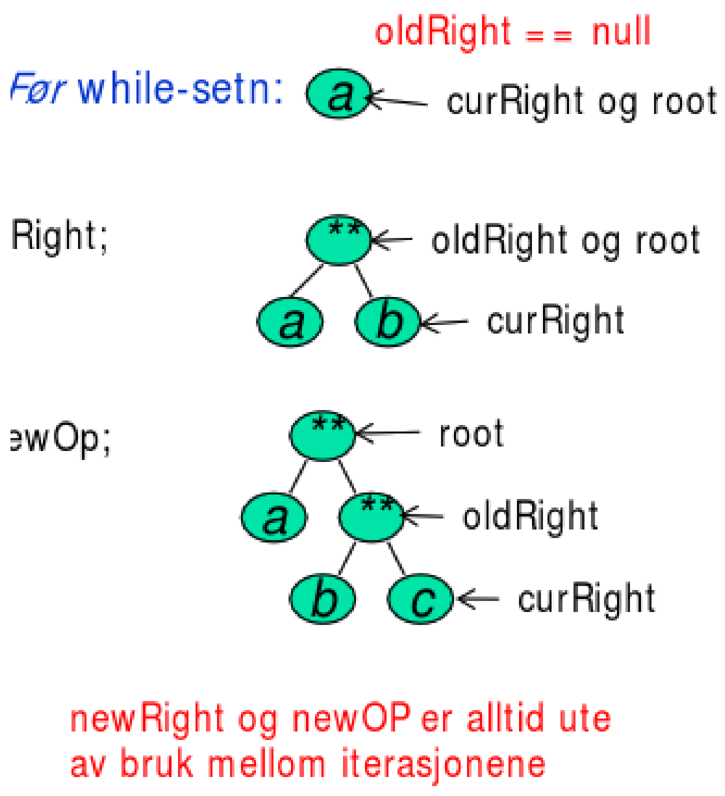
And we need to build them like that (from left to right):

```
                               e3  <- old right
                              /  \
e4        e3                 e4   e3
|        /  \                |   /  \
n       e4  e4               n  e4  e4   <- current right
        |   |                   |   |
        n   n                   n   n
```

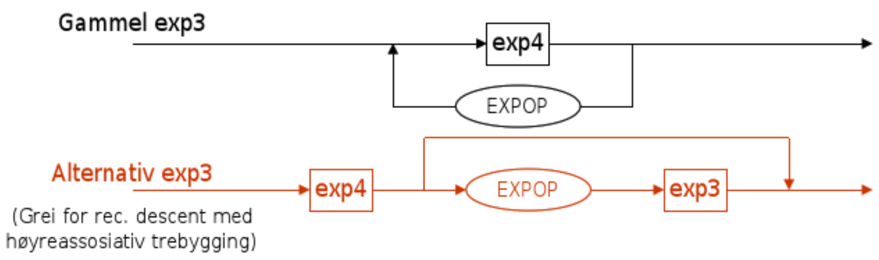
```
procedure exp3() : TreeNode {
  TreeNode oldRight, curRight, newRight, newOp;

  curRight = exp4();                // parsed first but remembered
  root = curRight; oldRight = null;

  while token = EOP do {            // see how many exponentiations there are
   getToken();
   newRight = exp4();
   newOp = newOpNode("**");
   newOp.right = newRight;
   if (oldRight == null) {
       newOp.left = curRight;
       root =  newOp;
   } else {
       newOp.left      = oldRight.right;
       oldRight.right = newOp;
   }
   oldRight = newOp;
   curRight = newRight;
  }
  return root;
}
```

The following sketches the build of the AST for the following expression

a∗∗**b**∗∗**c**



The earlier second subtask gave syntax diagrams as a "graphical" way to represent grammar productions. They may be helpful as inspiration for writing recursive descent parsers (or get an overview over a grammar in general), but they were not really helpful in solving the "implement ↑ in a right-associative way" question. We could rearrange the syntax diagram for exp_3 slightly in the following way:



That may more directly give inspiration to an implementation, looking as follows:

Listing 2: Alternative for right-assoc. ↑, no tree-building

```
procedure exp3() {
  exp4();
  if token = EXPOP then {
     getToken();
     exp3();
  }
}
```

When building the AST which doing the recursive descent, the code looks as follows

Listing 3: Alternative for right-assoc. ↑, *with* tree-building

```
procedure exp3() : TreeNode {
  TreeNode root; OpNode opNode;
  root := exp4();
  if token = EXPOP then {
    getToken();
    newRight := exp3()
    root := newOpnode("**", root, newRight);
  }
  return root
}
```

3. (left factorization) The starting point is the grammar from equation (2) on page 6. With the techniques from the lecture for "cleaning up" left-recursion we can transform it to

$$\begin{array}{rcl}
exp & \to & exp_1\; exp' \\
exp' & \to & \textbf{relop}\; exp_1 \;\mid\; \epsilon \\
exp_1 & \to & exp_2\; exp'_1 \\
exp'_1 & \to & \textbf{addop}\; exp_2\; exp'_1 \;\mid\; \epsilon \\
exp_2 & \to & exp_3\; exp'_2 \\
exp'_2 & \to & \textbf{mulop}\; exp_3\; exp'_2 \;\mid\; \epsilon \\
exp_3 & \to & exp_3\; exp'_3 \\
exp'_3 & \to & \textbf{expop}\; exp_3 \;\mid\; \epsilon \\
exp_4 & \to & \textbf{(}\; exp\; \textbf{)} \;\mid\; \textbf{number}
\end{array}$$

The fact that we keep unambiguity is not easy to see directly, but follows once we have done the next subtask, where we do the LL(1) check via looking at the first- and follow-sets

4. LL(1) check.

	First	*Follow*
exp	**number**, **(**	**\$**, **)**
exp'	ϵ, **relop**	**\$**, **)**
exp_1	**number**, **(**	**\$**, **)**, **relop**
exp'_1	ϵ, **addop**	**\$**, **)**, **relop**
exp_2	**number**, **(**	**\$**, **)**, **relop**, **addop**
exp'_2	ϵ, **mulop**	**\$**, **)**, **relop**, **addop**
exp_3	**number**, **(**	**\$**, **)**, **relop**, **addop**, **mulop**
exp'_3	ϵ, **expop**	**\$**, **)**, **relop**, **addop**, **mulop**
exp_4	**number**, **(**	**\$**, **)**, **relop**, **addop**, **mulop**, **expop**

	relop	**addop**	**mulop**	**expop**	**number**	**(**	**)**	**\$**
exp					$exp_1\; exp'$	$exp_1\; exp'$		
exp'	**relop** exp_1						ϵ	ϵ
exp_1					$exp_2\; exp'_1$	$exp_2\; exp'_1$		
exp'_1	ϵ	**addop** $exp_2\; exp'_1$					ϵ	ϵ
exp_2					$exp_3\; exp'_2$	$exp_3\; exp'_2$		
exp'_2	ϵ	ϵ	**mulop** $exp_3\; exp'_2$				ϵ	ϵ
exp_3					$exp_4\; exp'_3$	$exp_4\; exp'_3$		
exp'_3	ϵ	ϵ	ϵ	**expop** exp_3			ϵ	ϵ
exp_4					**number**	**(** exp **)**		

A better solution would probably be along the lines we have seen in the lecture...

$\square$