Universitetet i Oslo
Institutt for Informatikk

Reliable Systems
Martin Steffen, Gianluca Turin

INF 5110: Compiler construction

Spring 2021 — **Series 3** — 12. 3. 2021

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

Issued: 12. 3. 2021

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 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 perhaps in general, 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 derivation. Very 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 $\alpha_1 \neq \alpha_2$:

$$First_1(\alpha_1) \cap First_1(\alpha_2) = \emptyset \ .$$

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 $\alpha_1 \neq \alpha_2$:

$$First_1(\alpha_1 Follow_1(A)) \cap First_1(\alpha_2 Follow_1(A)) = \emptyset \ .$$

Be carefull with the definition. It's

$$First(\alpha_1 \ Follow(A)), \quad not \quad First(\alpha_1) \ Follow(A) \ .$$

Note: Lemma 2 from this exercise was *not* explicitly shown on the slides (the simpler Lemma 1 was)! 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, of course, directly embodies the principles of the lemma.

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

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

	First	*Follow*
S	$\boldsymbol{\epsilon}$, (	**$**,)

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):

	(	)	**$**
S	$S \to (\, S \,) \, S$	$S \to \boldsymbol{\epsilon}$	$S \to \boldsymbol{\epsilon}$

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 corresond to an error situation, but that would not indicate ambiguity.

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

[2] Don't forget the $\boldsymbol{\epsilon}$ 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.

```
1 procedure S() {
2     if token = "("
3     then getToken();
4          S()
5          match (")");
6          S()
7     else skip
8     end
9 end
```

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

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

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	$\epsilon, ($	$\$, (,)$

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.

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

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, which is the addition. As required, it's supposed to be left-associative, so we could add a new non-terminal, say exp', and reformulate the grammar as follows:

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

This was a solution proposed in some earlier semesters. On closer inspection, though, it turns out that this grammar is still ambiguous! As an example, take the following expression

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

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, of course. 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.

One could try to solve that dangling-plus similar the same way the dangling-else problem is solved , but we leave it at that. Alternatively, we can leave it to the parser (for instance to prefer shift over reduce for LR-parsers, similarly for LL(1) parsers) to disambiguate the situation.

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

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.

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

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. There were 3 precendence levels (and we had, in that earlier exercise, 3 different versions of expressions: original expressions, terms, and factors).

   Note: we have 4 levels of precedences, which gives *5* different "kinds" of expressions.

   Note also: the question is kind of not well-formulated. It is not clear what is meant 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]

   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 $\uparrow$

[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.

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

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

$$1 = 1 = 1 = 1$$

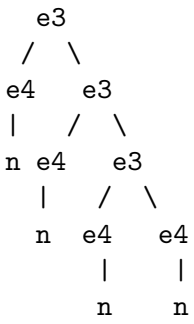
cannot be parsed (which was possible in the original grammar). It should be noted: to forbid such syntax is in-line with the requirements from the compila language from the *oblig*, where it's specified that such relational expressions are considered as syntactically disallowed. Perhaps one could make 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.

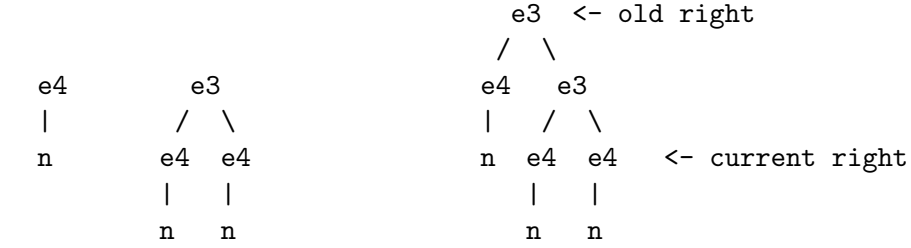
Besides the precedences: important is to understand how one gets the *associativity correct.*

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:



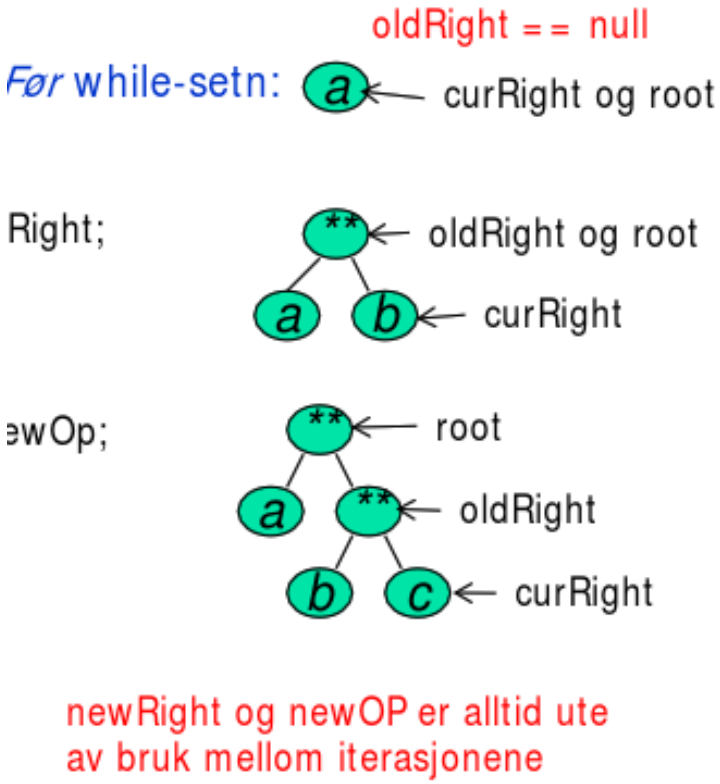
And we need to build them like that (from left to right):



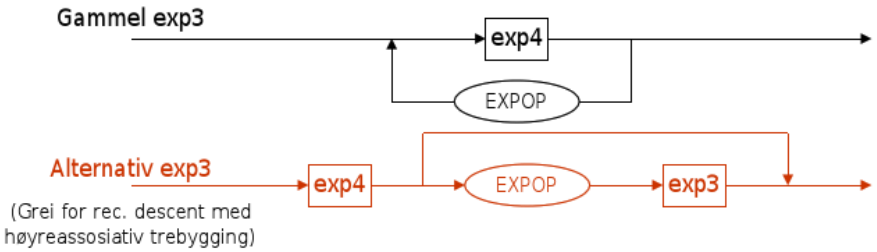

```
1  procedure exp3() : TreeNode {
2    TreeNode oldRight, curRight, newRight, newOp;
3
4    curRight = exp4();                // parsed first but remembered
5    root = curRight; oldRight = null;
6
7    while token = EOP do {            // see how many exponentiations there are
8     getToken();
9     newRight = exp4();
10    newOp = newOpNode("**");
11    newOp.right = newRight;
12    if (oldRight == null) {
13        newOp.left = curRight;
14        root =  newOp;
15    } else {
16        newOp.left      = oldRight.right;
17        oldRight.right = newOp;
18    }
19    oldRight = newOp;
20    curRight = newRight;
21   }
22   return root;
23 }
```

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

```
1 procedure exp3() {
2   exp4();
3   if token = EXPOP then {
4     getToken();
5     exp3();
6   }
7 }
```

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

Listing 2: Alternative for right-assoc. ↑, *with* tree-building

```
1  procedure exp3() : TreeNode {
2    TreeNode root; OpNode opNode;
3    root := exp4();
4    if token = EXPOP then {
5      getToken();
6      newRight := exp3()
7      root := newOpnode("**", root, newRight);
8    }
9    return root
10 }
```

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

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

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...

□