Universitetet i Oslo
Institutt for Informatikk
Reliable Systems
Martin Steffen

INF 5110: Compiler construction

Spring 2023 **Series 2** 9. 2. 2023

Topic: Context free grammars (Exercises with hints for solution)

Issued: 9. 2. 2023

This exercise set covers more than one lecture. It's about grammars, and partly for the lectures about *parsing*. We might not be able to cover it within 2 hours.

Exercise 1 (First- and follow sets) Compute the *First* and *Follow*-sets for the grammar Figure 1.

$$
\begin{array}{rcl}
\mathit{exp} & \rightarrow & \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' & \rightarrow & \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \boldsymbol{\epsilon} \\
\mathit{addop} & \rightarrow & \mathbf{+} \mid \mathbf{-} \\
\mathit{term} & \rightarrow & \mathit{factor}\ \mathit{term}' \\
\mathit{term}' & \rightarrow & \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \boldsymbol{\epsilon} \\
\mathit{mulop} & \rightarrow & \mathbf{*} \\
\mathit{factor} & \rightarrow & \mathbf{(}\ \mathit{exp}\ \mathbf{)} \mid \mathbf{number}
\end{array}
$$

Figure 1: Expression grammar (left-recursion removed)

Solution:

Listing 1: First sets

```
for all X ∈ Σ_T ∪ {ε} do
    First[X] := {X}
end;

for all non-terminals A do
   First[A] := {}
end
while there are changes to any First[A] do
   for each production A → X_1 ... X_n do
     k := 1;
     continue := true
     while continue = true and k ≤ n do
        First[A] := First[A] ∪ First[X_k] \ {ε}
        if ε ∉ First[X_k] then continue := false
        k := k + 1
     end;
     if   continue = true
```

```
    then  First[A] := First[A] ∪ {ϵ}
  end;
end
```

We have learnt algorithms to do that. They are repeated in this exercise for easy reference in Listing 1 and 2.

Listing 2: Follow sets

```
Follow[S] := {$}
for all non-terminals A ≠ S do
  Follow[A] := {}
end
while there are changes to any Follow-set do
  for each production A → X_1 ... X_n do
    for each X_i which is a non-terminal do
      Follow[X_i] := Follow[X_i] ∪ (First(X_{i+1} ... X_n) \ {ϵ})
      if ϵ ∈ First(X_{i+1} X_{i+2} ... X_n)
      then Follow[X_i] := Follow[X_i] ∪ Follow[A]
    end
  end
end
```

But one can also informally try to figure it out (at the danger that one forgets some symbols, especially when dealing with nullable symbols and **ϵ**-productions, one has to watch out for those). In any case, the first-sets are simpler. Furthermore, the definition of the follow sets *depends* on determining the first-sets. Therefore, in such an exercise, one *always* starts with the first-sets, and only then attempts the follow-sets.

For the first sets: the main complication are *nullable symbols,* and the grammar has those, so we need to watch out.

In the "recursive definition" of the algorithm for the first sets one should notice that the definition defines the first set not just for non-terminals, but for terminals and **ϵ**, as well!

Here's a "run" for the first-sets (the non-terminals are abbreviated in the obvious manner to save space). Pass 4 is not shown, nothing would change there compared to pass 3 and the algo finishes.

non-term	pass 1	pass 2	pass 3
e			(, **number**
e'		$\boldsymbol{\epsilon}, +, -$	
e'	$\{\boldsymbol{\epsilon}\}$	$\boldsymbol{\epsilon}, +, -$	
a	$+$		
a	$+, -$		
t		(, **number**	
t'		$\boldsymbol{\epsilon}, *$	
t'	$\boldsymbol{\epsilon}$	$\boldsymbol{\epsilon}, *$	
m	$*$		
f	(		
f	(, **number**		

Table 1: "Run" of the first-set algo

Here's the "collapse" of the result, i.e., that's the end result.

non-term	*First*
exp	(, **number**
exp'	$\epsilon, +, -$
addop	$+, -$
term	(, **number**
term'	$\epsilon, *$
mulop	$*$
factor	(, **number**

As a side remark: In the result, the primed non-terminals *term'* and *exp'* are the ones containing ϵ in the first set, indicating that they are nullable. That's not a coincidence. The grammar we are dealing with is (one variant of) the expression grammar, namely one on which the algo for left recursion removal has been applied. The algo introduces new terminals with appropriate rules. By convention and to be a bit systematic, for a non-terminals A with left recursion, a new non-terminal A' is added decorated with a prime, and the corresponding massaged rules contain an ϵ-production. In the current example, the new symbols are *term'* and *exp'*.

As another side remark: the table arranges the productions of the grammar in the order as appearing in the grammar. It's also assumed in the shown solution, that algo goes through each production in the order as shown in the table and in the grammar. That's not a necessity, the sketched algorithm formulates the loop-construct for one pass as "for each production do ...", it does not require a particular order. Actually, the order illustrated in the table is not the best one, treating the productions in reverse order would lead to quite faster termination.

So far for the first sets. The algo for the *follow sets* is similar in spirit to the first-set calculation, it's likewise a "saturation" algorithm. But the "table" used to simulate the run is organized slightly differently, because the slots now correspond to the *right-hand sides* of the productions, not the left-hand sides alone. The *initialization* of the algo start by having $\mathit{Follow}(\mathit{exp}) = \{\$\}$, all the rest is the empty set. As a general rule, we never add any ϵ to the follow-set, they simply don't belong there. However, we need to check if ϵ is in the considered *First*-set, as that plays a role.

We should also remember that we need in the construction not directly $\mathit{First}(X)$ but (corresponding) definition for sequences of symbols $\mathit{First}(X_1 \ldots X_k)$. By "corresponding" it is meant that "in spirit", it's the same definition, only applied to sequences. The lecture provides the definition of the first set of a word (but not in pseudo code form). Basically it's an iterated application of the definition of the first-set for *one* symbol (and taking into account nullable symbols in the analogous way that the definition/code of first-sets of a symbol treats that already (when dealing with the right-hand side of a production). So it's nothing really new.

In the table, **F** stands for the follow set. I write also here += for the operation increasing the current value of a given set (representing `Follow[X] := Follow[X]` $\cup\{\ldots\}$).

In the productions on the right-most column, I indicate by color the longest "post-fix" of the right-hand side which is nullable (that can be seen from the result of the first-sets). That's helpful, because those situations require special treatment by the algorithm. For example, in the situation

$$\mathit{exp}' \rightarrow \mathit{addop}\ \mathit{term}\ \mathit{exp}' \ ,$$

exp' is nullable, but the longer *term exp'* is not.[1] To treat the non-terminal *addop* in the corresponding case is easier as the postfix after it is not nullable. To treat the non-terminal *term* is slightly more complex, as the *exp'* is nullable. One should not forget: to treat *exp'* leads to a post-fix of ϵ, which also counts as nullable, therefore the "special treatment" applies to *exp'*, as well.

[1]Neither is *addop term exp'*, of course.

production	init	pass 1	pass 2
$\mathit{exp} \rightarrow \mathit{term}\ \mathit{exp}'$	$\mathbf{\$}$	$\mathbf{F}(\mathit{term}) = \{+,-\} \cup \{\mathbf{\$}\}$	$\mathbf{F}(\mathit{term}) \mathrel{+}= \{)\}$
		$\mathbf{F}(\mathit{exp}') = \{\mathbf{\$}\}$	$\mathbf{F}(\mathit{exp}') \mathrel{+}= \{)\}$
$\mathit{exp}' \rightarrow \mathit{addop}\ \mathit{term}\ \mathit{exp}'$		$\mathbf{F}(\mathit{addop}) = \{(,\mathbf{number}\}$	
		$\mathbf{F}(\mathit{term}) = \{+,-,\mathbf{\$}\} \cup \{\mathbf{\$}\}$	
		$\mathbf{F}(\mathit{exp}') = \mathbf{F}(\mathit{exp}')$	
$\mathit{exp}' \rightarrow \boldsymbol{\epsilon}$		$\mathbf{F}(\boldsymbol{\epsilon}) =$ not a case	
$\mathit{addop} \rightarrow +$		$+$ is a terminal, nothing's done	
$\mathit{addop} \rightarrow -$		same	
$\mathit{term} \rightarrow \mathit{factor}\ \mathit{term}'$		$\mathbf{F}(\mathit{factor}) = \{*\} \cup \mathbf{F}(\mathit{term})$	$\mathbf{F}(\mathit{factor}) \mathrel{+}= \{)\}$
		$\mathbf{F}(\mathit{term}') = \mathbf{F}(\mathit{term}) = \{\mathbf{\$},+,-\}$	$\mathbf{F}(\mathit{term}') \mathrel{+}= \{)\}$
$\mathit{term}' \rightarrow \mathit{mulop}\ \mathit{factor}\ \mathit{term}'$		$\mathbf{F}(\mathit{mulop}) = \{(,\mathbf{number}\}$	
		$\mathbf{F}(\mathit{factor}) = \{*\} \cup \mathbf{F}(\mathit{term}') = \{*,+,-,\mathbf{\$}\}$	
		$\mathbf{F}(\mathit{term}') = \text{\_}$	
$\mathit{term}' \rightarrow \boldsymbol{\epsilon}$		not a case	
$\mathit{mulop} \rightarrow *$		$*$ is a terminal	
$\mathit{factor} \rightarrow (\ \mathit{exp}\ )$		$\mathbf{F}(\mathit{exp}) = \{)\}$	
$\mathit{factor} \rightarrow \mathbf{number}$		terminal	

The "run" of the algo resp. the table used to represent the run shows the difference in the structure of the follow-calculation. In contrast to the first-algorithm, there there are 3 loops. The outer loop, which correspond to the colums (the "passes"). The second loop going through all grammar productions, and finally, for each production, the inner loop going through the symbols on the right-hand side one by one.

non-term	*Follow*
exp	$\mathbf{\$},)$
exp'	$\mathbf{\$},)$
addop	$(, \mathbf{number}$
term	$\mathbf{\$},), +, -$
term'	$\mathbf{\$},), +, -$
mulop	$(, \mathbf{number}$
factor	$\mathbf{\$},), +, -, *$

Table 2: Result: follow sets

□

Exercise 2 (Nullable) Describe an algorithm that finds all nullable non-terminals without first finding the first-sets.

Solution: It should be clear that there *is* an algo, already from the fact that in the lecture we discussed one (in different representations). Apart from that: taking the original definition of nullability: from that one it's not immediately clear, because the definition states a condition like

$$A \Rightarrow^* \boldsymbol{\epsilon}\ .$$

However, given a grammar, one can intuitively make a "graph search", taking a non-terminal, and then look at "chains of productions" (which corresponds to a graph traversal) and see if one hits on $\boldsymbol{\epsilon}$ without passing through a terminal. Doing that for all non-terminals would answer the question.

That's not the smartest way to do it, however, and that's not the way we handled the first- and follow-sets algorithmically (perhaps also compare the ϵ-closure). Very generally, one is better off to calculate "simultaneously" the first- and follow-sets, resp. in the task *here*, calculate the question of nullability simultaneously for all non-terminals (not one by one). That may (very roughly) be compared to, for instance the idea behind Dijsktra's shortest-path algo (covered for instance in INF2220 (old numbering)): Also that algo works calculating shortest path *for all pairs of nodes* not simply for the one particular pair one might be interested in (and iterate that for all pairs).

Anyhow: a good solution to the task here is to do better than the nullability-one-by-one solution.

One way of doing it is: we arrange for a data structure nullable, which is a set of non-terminals (perhaps concretely some collection data structure), containing all non-terms of the given grammar which are nullable. To be more precise: that set contains all nullable non-terms *when the algo terminates.* During the run of the algo, it contains "the current knowledge or estimation" which of the non-terms are already known to be nullable. It's a crucial characteristic of this kind of algorithm (like the follow/first calculations as well, and many others) that the "estimation grows only in one direction", meaning that during the run of the algo, the current knowledge of nullable symbols, i.e, the corresponding set data structure, only *grows*. In each stage (or "pass" or iteration), the algo potentially *adds* further symbols, when detecting nullability of further symbols so far not (yet) known to be nullable. The algo terminates, if now *new* symbols are detected meaning basically that the set of nullable symbols does not *grow* any more (it *stabilizes*).

The *input* of the algo is a grammar in BNF. Output is the set of nullable non-terminal symbols, kept in, say *nullable*. The general structure is (as for the first and follow algos,and similar ones):

```
initialize nullable
while not stabilized
  incease nullable (looking at the grammar, production after production)
end
return nullable
```

It is important to realize: going through the grammar "production after production" (or "terminal after terminal" or whatever) does *not* mean, that, after having done one sweep through all the productions, one is done. That is typically not the case. One stops if, going *repeatedly* through the productions, it turns out that no new info can be learnt ("stabilization", "saturation"), *then* we terminate. See also how we filled in the tables with the "rounds" or "passes" when calculating the first- and follow-sets.

1. Initialization: start with the empty set.
2. Repeat, until no more elements are added: if there's a non-terminal A with

$$A \to \ldots B_1 B_2 \ldots B_n$$

   where *all* B_i are already (at the current stage) members of *nullable*, then add A to *nullable*.

Note that a rule like $A \to \boldsymbol{\epsilon}$ is covered for n but the recipe in the body of the loop, for $n = 0$.

Alternatively, more in line with the data structure of the first-algo: we could arrange the data structure in such a way that one has an boolean *array*, indexed by the non-terminals. It's of course the same. □

Exercise 3 (Associativity and precedence) Take the binary ops $+$, $-$, $*$, $/$ and $\uparrow$. Let's agree also on the following precedences and associativity

op	precedence	associativity
$+, -$	low	left assoc.
$*, /$	higher	left. assoc.
$\uparrow$	highest	right. assoc

Write an *unambiguous* grammar that captures the given precedences and associativies (of course, directly with a BNF grammar, without allowing yourself specifying those requirements as extra side-conditions).

Solution: We have learned in the lecture how it works, at least for the operators *except* the exponentiation.

The "flat grammar", simple, elegant, but with utter disrespect for associativity and precedence could look as follows (it's not required for the task):

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

The ok grammar (without exponentiation) presented in the lecture and script looked as follows

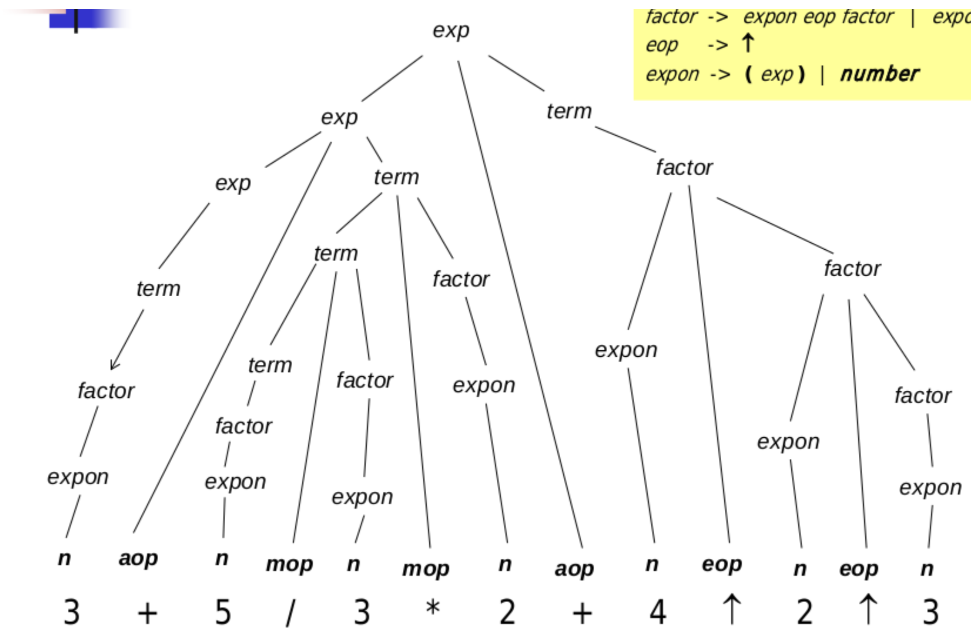
$$\begin{array}{rcl} \mathit{exp} & \to & \mathit{exp}\ \mathit{addop}\ \mathit{term} \mid \mathit{term} \\ \mathit{addop} & \to & + \mid - \\ \mathit{term} & \to & \mathit{term}\ \mathit{mulop}\ \mathit{factor} \mid \mathit{factor} \\ \mathit{mulop} & \to & * \\ \mathit{factor} & \to & (\, \mathit{exp}\,) \mid \textbf{number} \end{array}$$

Now we have to include the exponentiation. In the same way we did for the terms and factors: we need a new non-terminal, here say *expon*, and furthermore, we need to get it right-associative. Therefore, the rule for *factor* is formulated using *right-recursion:* the *factor* occurs on the right, not on the left (as for *term* and *exp*)

$$\begin{array}{rcl} \mathit{exp} & \to & \mathit{exp}\ \mathit{addop}\ \mathit{term} \mid \mathit{term} \\ \mathit{addop} & \to & + \mid - \\ \mathit{term} & \to & \mathit{term}\ \mathit{mulop}\ \mathit{term} \mid \mathit{factor} \\ \mathit{mulop} & \to & * \\ \mathit{factor} & \to & \mathit{expon}\ \mathit{eop}\ \mathit{factor} \mid \mathit{expon} \\ \mathit{eop} & \to & \uparrow \\ \mathit{expon} & \to & (\, \mathit{exp}\,) \mid \textbf{number} \end{array}$$

Here's an example CST (concrete syntax tree) for the expression.

$$3 + 5/3 * 2 + 4 \uparrow 2 \uparrow 3$$



□

Exercise 4 (Tiny grammar) For the grammar given answer the following questions:

- Is the grammar *unambiguious?*
- How can we change the grammar, so that TINY allows empty statements?
- How can we arrange it that semicolons are required *in between* statements, not *after* statements?
- What's the precedence and associativity of the different operators?

program	→	*stmts*
stmts	→	*stmts* **;** *stmt* | *stmt*
stmt	→	*if-stmt* | *repeat-stmt* | *assign-stmt*
	|	*read-stmt* | *write-stmt*
if-stmt	→	**if** *expr* **then** *stmt* **end**
	|	**if** *expr* **then** *stmt* **else** *stmt* **end**
repeat-stmt	→	**repeat** *stmts* **until** *expr*
assign-stmt	→	**identifier :=** *expr*
read-stmt	→	**read identifier**
write-stmt	→	**write** *expr*
expr	→	*simple-expr comparison-op simple-expr* | *simple-expr*
comparison-op	→	$<$ | $=$
simple-expr	→	*simple-expr addop term* | *term*
addop	→	$+$ | $-$
term	→	*term mulop factor* | *factor*
mulop	→	$*$ | $/$
factor	→	**(** *expr* **)** | **number** | **identifier**

Solution: [of Exercise 4]

ambiguity: One might be aware that many properties of context-free grammar (not to mention context-free languages ...) are "hard", even undecidable. Whether or not a grammar is unambiguous is one of them: that is in general undecidable (and that immediately makes also the question if a context free language has an unambiguous grammar undecidable). Of course, for this particular grammar the question can be expected to have a definite answer (if only for the reason that otherwise the question would be meaninglessly hard ...). Still, those remarks about *general* undecidability could warn the reader that ambiguity of grammars is not a trivial matter (and in particular there's no algorithm that definitely can check the issue). That also might imply that showing that this particular grammar *is* unambiguous (should that be the case) is not easy: we would have to find an argument, that each word of terminals of the TINY-language has a unique parse tree, and there are infinitely many. That sounds hard (especially since we have not covered that question in the lecture). It seems more easy, should that be the case, to prove that the grammar is *ambiguous.* All we need is *one* word with 2 different parse trees. However, this general "smart" meta-consideration will lead us down the wrong path right here, the answer will be that the grammar seems unambiguous (but we won't make an attempt in proving it).

That makes it a priori plausible that the given grammar is ambiguous, we only are requested to nail down at least one "part" which causes ambiguity (there may be more than one reason, but we only need to find one).

Before doing so, and as a side remark: there are doable approaches which allows (indirectly) to prove unambiguity in some cases: for instance, if one could show that a given grammar is LL(k) or LR(k) or one of such classes, then we indirectly know the grammar is *unambiguous.*

Now: what's the reason for ambiguity? There might be the "usual suspects". Since the grammar is for a "real" (toy) language and not an arbitrary artificial grammar, we should concentrate on those usual suspects.

As we learned, a common source of ambiguity in grammars of programming languages is that, especially for binary constructs, the "grouping" is not fixed by the grammar (which then gives ambiuity for the parse tree). Typically that involves question of "associativity" or "precedence" (as discussed with the previous exercise). Another example we had, though not with plain binary operators but related nonetheless, is the phenomenon of "dangling else".

The dangling else is not a problem here.

Now, what about associativity. The grammar seem to make use of the techniques we used in the lecture, splitting up expressions into factors and terms. So, that part takes care of precedence and associativity of multiplication, addition, etc.

Now, remains the suspicious comparison expressions (comparing two simple expressions). At first sight that might look having problems with associativity. However: it would be ambiguous if we had a production for expressions like

$$\mathit{exp} \rightarrow \mathit{exp}\ \mathit{comparison\text{-}op}\ \mathit{exp} \tag{1}$$

or the simple expressions would be done like that:

$$\mathit{simple\text{-}expr} \rightarrow \mathit{simple\text{-}expr}\ \mathit{comparison\text{-}op}\ \mathit{simple\text{-}expr} \tag{2}$$

or similar arrangements. would lead to ambiguity.

So, since the grammar is not of any of the forms, it is very plausible that it's *unambiguous.*

Additional remarks: If one had an different version, for instance with a production of the form of equation (1) or (2), and thus the answer would be that the grammar is indeed ambiguous, in that case a good answer would give one example of an expression and give 2 different parse trees for that.

As a rather esoteric side remark: Technically, the question actually is not whether *parts* of the grammar are ambiguous (for instance the part starting with the expression non-terminal in the changed version), but if the *overall* grammar is ambiguous. So in principle, the grammar could be unambigous, even if expressions were parsed ambiguously, namely in the (weird) case when expressions can never be derived from the start symbol. Such grammars with unreachable parts or parts that can never be derived into a word of terminals are obviously “defective” in that there should never be “useless” symbols or productions in a grammar.

Anyway, the TINY grammar (and all decent grammars) does not suffer from such defects. Given an ambiguous grammar, pointing out and explaining the ambiguity (or lack thereof) for expressions is an answer good enough.

Empty statements: That’s trivial, we simply add *stmt* $\rightarrow \boldsymbol{\epsilon}$. This will make it possible to write statement sequences like **; ; ; ; ;**.

Semicolon as terminator, not separator: We could do

$$stmts \rightarrow stmts\ stmt\ \mathbf{;}\ \mid\ stmt\ \mathbf{;}$$

See also the language specification for the oblig, where the grammar is written in EBNF, not BNF.

Associativity and precedence:

op	precedence	associativity
$*, /$	highest	left
$+, -$	medium	left
$<, =$	lowest	non-associative

We can also think of semicolon and its associativity. Except that it’s not really counted among operators, typically.

Associativity and precedences for expressions and statements becomes more tricky if one deals with languages which “mixes” them. For instance, as is C-like languages, that every statement is *also* an expression. Then we have to think of

```
5 + x = 5
```

If for instance in Java

```java
public class Stmtasexpr {
    static int x = 23;
    public static void main(String[] arg) {
        //        x= 5 + x = 5;
        x = 5 + (x = 5);
    }
}
```

□