

COS341 Practical 2: Academic Year 2023

Note: This P#2 can be solved *without* having solved the foregoing P#1.

GIVEN is the following **context-free grammar**.

Note: To make life easier for students, *most lexical tokens* are no longer than only *one* character and all semantic type-correctness is already *hard-coded into the syntax* of the grammar's language.

The Students' Programming Language "SPL"

PROGR	::=	ALGO PROCDEFS	
PROCDEFS	::=	, PROC PROCDEFS	
PROCDEFS	::= ϵ		
PROC	::=	p DIGITS { PROGR } // <i>Proc can possibly have further inner Proc-Defs!</i>	
DIGITS	::=	D MORE	
D	::=	0 1 2 3 4 5 6 7 8 9	
MORE	::=	DIGITS	
MORE	::= ϵ		
ALGO	::=	INSTR COMMENT SEQ	
SEQ	::=	; ALGO	
SEQ	::= ϵ		
INSTR	::=	INPUT // <i>From console-user</i>	
INSTR	::=	OUTPUT // <i>To screen</i>	
INSTR	::=	ASSIGN	
INSTR	::=	CALL	
INSTR	::=	LOOP	
INSTR	::=	BRANCH	
INSTR	::=	h // <i>halt</i>	
CALL	::=	c p DIGITS // <i>The c stands for call (whereby p DIGITS is a Proc-Name)</i>	
ASSIGN	::=	NUMVAR := NUMEXPR // <i>Type-correctness is hard-coded in the grammar</i>	
ASSIGN	::=	BOOLVAR := BOOLEXP	
ASSIGN	::=	STRINGV := STRI	
LOOP	::=	w (BOOLEXP){ ALGO } // <i>The w stands for while</i>	
BRANCH	::=	i (BOOLEXP) t { ALGO } ELSE // <i>The i stands for if, the t stands for then</i>	
ELSE	::=	e { ALGO } // <i>The e stands for else</i>	
ELSE	::= ϵ		
NUMVAR	::=	n DIGITS // <i>Type n, for numeric, is already hard-coded into the syntax</i>	
BOOLVAR	::=	b DIGITS // <i>Type b, for boolean, is already hard-coded into the syntax</i>	
STRINGV	::=	s DIGITS // <i>Type s, for stringish, is already hard-coded into the syntax</i>	
NUMEXPR	::=	a (NUMEXPR , NUMEXPR) // <i>The a stands for addition</i>	
NUMEXPR	::=	m (NUMEXPR , NUMEXPR) // <i>The m stands for multiplication</i>	
NUMEXPR	::=	d (NUMEXPR , NUMEXPR) // <i>The d stands for division</i>	
NUMEXPR	::=	NUMVAR	
NUMEXPR	::=	DECNUM // <i>Decimal number with maximally two Digits behind the dot</i>	

DECNUM	::=	0.00 NEG POS
NEG	::=	–POS
POS	::=	INT.DD // Two Digits behind the dot: Digit D as defined above
INT	::=	1MORE 2MORE 3MORE 4MORE ... 8MORE 9MORE
BOOLEXP	::=	LOGIC
BOOLEXP	::=	CMPR // Comparisons of numbers (with true or false as outcomes)
LOGIC	::=	BOOLVAR
LOGIC	::=	T F // True, False: the Boolean constants
LOGIC	::=	^ (BOOLEXP , BOOLEXP) // And-Conjunction
LOGIC	::=	v (BOOLEXP , BOOLEXP) // Or-Disjunction
LOGIC	::=	! (BOOLEXP) // Not-Negation
CMPR	::=	E (NUMEXP , NUMEXP) // Equality-Comparison for Numbers
CMPR	::=	< (NUMEXP , NUMEXP) // Lesser-Comparison for Numbers
CMPR	::=	> (NUMEXP , NUMEXP) // Larger-Comparison for Numbers
STRI	::=	"CCCCCCCCCCCCCCCC" // Short string of constant length 15
C	::=	all the usual ASCII keyboard characters, including the blank_space!
COMMENT	::=	*CCCCCCCCCCCCCCCC* // Short text of constant length 15
COMMENT	::=	ε // optional
INPUT	::=	g NUMVAR // The g stands for getting a number from the user
OUTPUT	::=	TEXT VALUE
VALUE	::=	o NUMVAR // The o stands for output
TEXT	::=	r STRINGV // The r stands for response (to the user)

Also note:

Blank symbols `_` as well as **new-line** pseudo-symbols should be *syntactically irrelevant* – in other words: They *may possibly* separate subsequent tokens from each other *for the sake of convenience* as "syntactic sugar"; however it should also be possible to analyse totally "dense" input strings in which no such symbols are present. The EXCEPTION is a blank symbol INSIDE a STRING where it *counts* as character!

Work-STEPS for Doing this Practical:

- Analyse with pen and paper whether the given grammar is *ambiguous*? If it is, then "deal with it" as explained in the book: either transform it into an equivalent non-ambiguous grammar, or somehow stipulate some meta-regulations for handling ambiguous situations.
- Analyse with pen and paper whether the given grammar is *LL1*? If it is, then enjoy the easy job of writing a **recursive top-down parser** :) (Otherwise your work will be somewhat more difficult...)
 - FOR THE "GEEKS":** If you figure out that the given grammar is "mostly" LL1, with only a few rules violating the LL1 property, then you can try an advanced solution with TWO inter-connected parser-components: One LL1 component for the LL1-compliant parts of the grammar, and a second more complicated parser-component only for those grammar-rules which are not LL1-compliant: This second component would then be called from within the LL1 parser-component as a kind of Exception-Handler. Note: This special technique is **not recommended for non-"geeks"**!
- If your previously analysis revealed that the grammar is not in LL1 then analyse further with pen and paper whether the grammar is *still in SLR*? If that is the case, then continue to work with our textbook's knowledge – otherwise you might perhaps have to resort to other (more "powerful") parsing algorithms which you can find in the scientific literature on the internet.

- Because you already know from theoretical computer science that *Regular Languages* are included in the *Context-free Languages*, you may **decide for yourself**
 - **whether you wish** to implement a *lexer-less parser* (which carries out lexical analysis and syntax analysis "all in one"), **or**
 - **whether you wish** to implement a *separate lexer sub-module* which "feeds" already pre-analysed tokens into a subsequent parser (which might consequently be somewhat "smaller" in program-size and of better parse-time-efficiency): **the choice is your's**: See the **Software Module Specification** shown below ↓
- In this Practical 2, which is related to Chapter 2 of our book, the **utilisation of any pre-existing parser-generators** (such as ANTLR or similar) is **NOT allowed**, because the **educational purpose** of this practical is the "hand-crafting" of the parser in order to **deeply understand** the parser's "inner working".

**YOUR SOFTWARE,
to be submitted:**

INPUT: un-structured text string →	<u>PARSER</u> Implementation Option A: Pure parser without Lexer <i>(mono-phased)</i> Implementation Option B: Lexer → [Tokens] → Parser <i>(bi-phased)</i>	→ OUTPUT: * Error Message IF Input was NOT in the Lanuage of the given context-free grammar * Abstract Syntax Tree of the input OTHERWISE
------------------------------------	--	--

Format of Input and Output:

The input will be an unstructured sequence of characters presented in a simple *.txt file. Your software must be able to open the text file to process the unstructured sequence of characters.

The output must be presented in an **XML file**, such that it can be both displayed by a browser as well as also be "saved for later". Thereby **each node** of the Abstract Syntax Tree has its own internal structure (similar to a "struct" in C++) which must also be captured in XML as follows:

Unique Node ID Number
Contents
List of Child-Nodes

Thereby,

- The **unique node ID** number will be used both for linking the nodes of the tree to each other (as in *List of Child Nodes*) as well as – later in a future practical – also to link the node to an external semantic data base (table) in which further information about that node will be kept.
- **Contents** is either a **terminal token** from the input language (if the node is a leaf node) or a **Non-Terminal Symbol** of the grammar (if the node is an inner node or the root).
- The *List of Child Nodes* is empty if the node is a leaf node (containing a **terminal token**); for an inner node or the root node (containing a Non-Terminal symbol) the List of Child Nodes is non-empty.

Also Note:

- As per **sub-section 2.16.3** of our book, which you must study before programming your software, the **Abstract Syntax Tree** has been *pruned* of all *un-necessary* symbols from the input-stream. "Un-necessary" means that those symbols are no longer needed to the "understanding" of the "meaning" of the input, because this *meaning has "migrated" into the branch-structure of the Tree itself*.
 - In assessing the quality of your work, **the tutors** will also (inter alia) keep an eye on *whether your Output Tree still contains any un-necessary concrete syntax symbols!*
- **The tutors will specify in further details** whether you must submit your Software as un-compiled open source code, or as a pre-compiled executable program, or in both forms.
- Moreover the **tutors will specify in further details** whether you must use a specific programming language to compose your Parser, or whether you are allowed to utilise any programming language of your own choice.

And now: **HAPPY CODING :)**