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 }  // Proc can possibly have further inner Proc-Defs!

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       // From console-user
INSTR     ::=   OUTPUT      // To screen
INSTR     ::=   ASSIGN
INSTR     ::=   CALL
INSTR     ::=   LOOP
INSTR     ::=   BRANCH
INSTR     ::=   h   // halt

CALL      ::=   c pDIGITS // The c stands for call (whereby p DIGITS is a Proc-Name)

ASSIGN    ::=   NUMVAR := NUMEXPR // Type-correctness is hard-coded in the grammar
ASSIGN    ::=   BOOLVAR := BOOLEXPR
ASSIGN    ::=   STRINGV  := STRI

LOOP      ::=   w(BOOLEXPR){ALGO}        // The w stands for while

BRANCH    ::=   i(BOOLEXPR)t{ALGO}ELSE // The i stands for if, the t stands for then

ELSE      ::=   e{ALGO}                  // The e stands for else
ELSE      ::= ε

NUMVAR    ::=   n DIGITS     // Type n, for numeric, is already hard-coded into the syntax
BOOLVAR   ::=   b DIGITS     // Type b, for boolean, is already hard-coded into the syntax
STRINGV   ::=   s DIGITS     // Type s, for stringish, is already hard-coded into the syntax

NUMEXPR   ::=   a ( NUMEXPR , NUMEXPR )      // The a stands for addition
NUMEXPR   ::=   m ( NUMEXPR , NUMEXPR )      // The m stands for multiplication
NUMEXPR   ::=   d ( NUMEXPR , NUMEXPR )      // The d stands for division
NUMEXPR   ::=   NUMVAR
NUMEXPR   ::=   DECNUM     // Decimal number with maximally two Digits behind the dot
```

Turn the page →

DECNUM ::= **0.00** | NEG | POS

NEG ::= **–**POS
POS ::= INT**.**DD *// Two Digits behind the dot: Digit D as defined above*
INT ::= **1**MORE | **2**MORE | **3**MORE | **4**MORE | ... | **8**MORE | **9**MORE

BOOLEXPR ::= LOGIC
BOOLEXPR ::= CMPR *// Comparisons of numbers (with true or false as outcomes)*

LOGIC ::= BOOLVAR
LOGIC ::= **T** | **F** *// True, False: the Boolean constants*
LOGIC ::= **^ (** BOOLEXPR **,** BOOLEXPR **)** *// And-Conjunction*
LOGIC ::= **v (** BOOLEXPR **,** BOOLEXPR **)** *// Or-Disjunction*
LOGIC ::= **! (** BOOLEXPR **)** *// Not-Negation*
CMPR ::= **E (** NUMEXPR **,** NUMEXPR **)** *// **E**quality-Comparison for Numbers*
CMPR ::= **<** **(** NUMEXPR **,** NUMEXPR **)** *// Lesser-Comparison for Numbers*
CMPR ::= **>** **(** NUMEXPR **,** NUMEXPR **)** *// Larger-Comparison for Numbers*

STRI ::= "CCCCCCCCCCCCCCC" *// **Short** string of **constant** length 15*
C ::= all the usual ASCII keyboard characters, including the blank_space!

COMMENT ::= ***CCCCCCCCCCCCCCC*** *// Short text of **constant length 15***
COMMENT ::= ε *// optional*

INPUT ::= **g** NUMVAR *// The **g** stands for **g**etting a number from the user*

OUTPUT ::= TEXT | VALUE
VALUE ::= **o** NUMVAR *// The **o** stands for **o**utput*
TEXT ::= **r** STRINGV *// The **r** stands for **r**esponse (to the user)*

Also note:
Blank symbols **_** as well as **new-line** pseudo-symbols should be s*yntactically 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 →	**PARSER** Implementation ***Option A:*** **Pure parser** *without* Lexer (*mono-phased*) Implementation ***Option B:*** **Lexer** → [Tokens] → **Parser** (*bi-phased*)	**→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 :)**