COS**341** Practical

Semester Project

Part A: Syntax Analysis

Important:

For any questions, when in need for clarification, **visit the Tutor personally** during his consultation hour in the Lab.
No remote/online/e-mail consultation will be provided**!**

Organisation

- The submission deadline – to be announced later – will be the same for all students
- Thereafter, the Lab-Tutor will start with the software-testing, which will take a considerable amount of time (because there are more than 100 students in the course)
- Before the submission deadline, you can ask the Lab-Tutor for some “general advice” (for example: “how he wants the output file to be formatted”), however you are not allowed to ask the Lab-Tutor for advice about how to solve the project-problem that **you** are supposed to solve alone by yourself.
- Each student must work ALONE. No group-work, and **no plagiarism!**

Input and Output

- Your software must be able to consume an ASCII ***.txt** file which contains a “raw” un-formatted string as input.
- Your software “consumes” this string, analyses it, and emits any of the following three outputs:
  - If the input string is lexically wrong (according to the lexer which is housed in your software) then a “LEXICAL ERROR” is emitted together with some information about where in the input string the lexer got stuck; (no parsing is started).
  - If the input string is lexically correct but grammatically wrong (according to the parser which is housed in your software) then a “SYNTAX ERROR” is emitted together with some information about where the parser got stuck; (no syntax tree file is produced).
  - If the input string is both lexically and grammatically correct, then the parse tree is emitted in well-structured textual form as an ***.XML**/*.HTML file that a web-browser can render; (the Tutor will inspect this output for marking).

HOW to work

- **Pen-and-Paper BEFORE software-programming!**
- For **Lexical Analysis**:
  - From NFA to MinDFA **with pen and paper**; thereafter implement the MinDFA
  - *You can decide* whether to re-start the MinDFA according to the “First Match” or the “Longest Match” strategy, as long as your software functions properly.
- For **Syntax Analysis**:
  - Analyse **with pen and paper** whether the given grammar (following slides) is ambiguous, (whereby you can substitute “long” tokens by simple characters such that your grammar looks like the simple a-b-c-grammars in the textbook)
    - IF it is, then transform it into an equivalent non-ambiguous grammar, without changing the language SPL!
  - Analyse **with pen and paper** whether the non-ambiguous grammar can be LL-parsed with a recursive descent (top-down) parser:
    - IF it is NOT, then you must implement a more complicated parser on the basis of a more advanced (not so simplistic) parsing algorithm.

This year the *Student Programming Language* (*SPL*) has a different grammar ***G*** such that the students cannot re-use last year's solutions ☺

- **S**PLProgr → ProcDefs **main** { Algorithm **halt** ; VarDecl }
- ProcDefs → *// nothing (nullable)*
- ProcDefs → PD **,** ProcDefs
- PD → **proc** userDefinedName { ProcDefs Algorithm **return** ; VarDecl }

*// **Generic token-class** from the Lexer!*
*// The **regular expression** is given "after" the grammar*

- Algorithm → *// nothing (nullable)*
- Algorithm → Instr ; Algorithm
- Instr → Assign
- Instr → Branch
- Instr → Loop
- Instr → PCall
- Assign → LHS := Expr
- Branch → **if** (Expr) **then** { Algorithm } Alternat
- Alternat → *// nothing (nullable)*
- Alternat → **else** { Algorithm }

- Loop → **do** { Algorithm } **until** (Expr)
- Loop → **while** (Expr) **do** { Algorithm }

- LHS → **output**
- LHS → Var
- LHS → Field

- Expr → Const
- Expr → Var
- Expr → Field
- Expr → UnOp
- Expr → BinOp

- PCall → **call** userDefinedName
- Var → userDefinedName
- Field → userDefinedName[Var]
- Field → userDefinedName[Const]

- Const → ShortString
- Const → Number
- Const → **true**
- Const → **false**

*// **Generic token-classes** from the Lexer!*
*// The **regular expression** is given "after" the grammar*

- UnOp → **input**(Var)
- UnOp → **not**(Expr)

- BinOp → **and**(Expr,Expr)
- BinOp → **or**(Expr,Expr)
- BinOp → **eq**(Expr,Expr)
- BinOp → **larger**(Expr,Expr)
- BinOp → **add**(Expr,Expr)
- BinOp → **sub**(Expr,Expr)
- BinOp → **mult**(Expr,Expr)

- VarDecl → *// nothing (nullable)*
- VarDecl → Dec ; VarDecl

- Dec → TYP Var
- Dec → **arr** TYP[Const] Var

- TYP → **num**
- TYP → **bool**
- TYP → **string**

Additional Remarks

- The given grammar ***permits*** the formulation of ***ill**-typed* SPL programs in which (for example) a ***string*** is ***subtracted*** from a ***truth-value*** which does not make any sense.
  - **Later**, in the Static-Semantics Practical (Project **Part B**), we emit *Type-Error* messages upon the detection of such ill-defined situations.
- This year's **SPL** Programs are somewhat peculiar in the sense that all variable declarations appear at the end of a program, whereas in the "real world" they appear at the beginning of a program ☺
  - It is part of your "learning-experience" to understand that new programming languages can be defined arbitrarily, and that it does not really matter where the variable declarations are placed ☺

For the **Lexer**

- A **blank_space**, *or* a **carriage_return**, may generally be assumed to be a **separator** of two different subsequent tokens: **Re-set MinDFA** to its starting state.
  - An *exception* to this general rule is a **blank_space** ***inside*** one ***"string token"*** **!**
- The *Regular Expressions* from which the MinDFA must be constructed (with pen and paper) are characterised as follows:
  - **All** blue **terminal symbols** (from the foregoing grammar) **are tokens** (from the Lexer's perspective)
  - For **numbers**, **strings**, and **userDefinedNames** ➔ see the following slides…

Numbers (null, or positive or negative integers)

- **Regular Expression** (with shorthand notation):

0 | (([1–9] | (–.[1–9])).[0–9]*)

userDefinedNames

[a–z].([a–z] | [0–9])*

- Explanation in words: *at least one small alphabetical character, possibly followed by any mixed sequence of digits and/or further alphabetical characters*
  - Additional remark: In a **later** practical (Project **Part B**) we will emit a *semantic error-message* when any userDefinedName is *identical* with a **keyword** of SPL, ***if*** such an error is ***not already*** detected by the parser.
    - For example, if a user names another procedure “**main**”, the parser will *surely* “stumble”, because the grammar itself allows for only one **main** to be present.

ShortStrings

- *Between quotation marks, **maximally fifteen** CAPITAL letters, or digits, or blank_space symbols*
  - **Examples:**
    - **“”**
    - **“ “**
    - **“HELLO”**
    - **“HI COS341 GUYS”**

- Please create the **Regular Expression**, as an exercise, by yourself.

Further Advice: Implementation

- Tokens produced by the lexer can usefully be **implemented** as *triples* [ID/class/contents], whereby:
  - **ID** is a unique identification number (position of the token in the stream).
  - **class** is the token's type (for example: number, or keyword) which corresponds to a terminal symbol in the grammar/syntax-tree: If you abbreviate the class by a single character then you basically obtain the simple "a-b-c"-Grammars which are shown in all the examples of the textbook (Chapter 2).
  - **contents** is the actual instance of the token-type. For example: if the type is *keyword*, then the contents could be *else* – or if the token-type is *number*, the contents could be *–736*
- Nodes in the syntax trees produced by the parser can be usefully **implemented** as *triples* [ID/class/contents], too, whereby:
  - **ID** is a unique node number (via which *we will later connect semantic information* to it in the symbol table, for example Type-Checking-Information : see ➔ Chapters 3 and 5).
  - **class** is the Non-Terminal Symbols for inner nodes, and the terminal symbol for leaf-nodes.
  - **contents** is a list of pointers to the child-nodes in the case of inner nodes, or a pointer to a Token-ID in the case of leaf-nodes.

And now…

- HAPPY WORKING!

  ☺

- The submission deadline will be announced separately via a ClickUp Message