

# COS341 2020

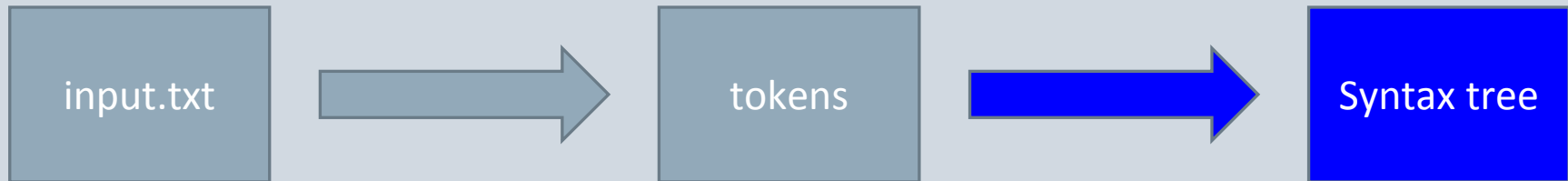
---

PROJECT Specification: **Task Booklet 01**

# Introduction

---

**Your task** is to implement a parser that can build a syntax tree from a list of tokens. This list of tokens will be created by your lexer from the previous part of this project.



# Grammar of SPL

---

The following slides define the grammar of SPL using productions, (similar to what is found in your textbook).

Productions will be of the form  $N \rightarrow X_1 \dots X_n$

- Where N is a **non-terminal** symbol.
- $X_1 \dots X_n$  are zero or more symbols which are either **terminal** or **non-terminal**.
  - *Terminal* symbols that are italic refer to **tokens** defined by regular expressions from the previous part of this project.

# Grammar of an SPL Program

---

PROG → CODE

PROG → CODE ; PROC\_DEFS

PROC\_DEFS → PROC

PROC\_DEFS → PROC PROC\_DEFS

PROC → *proc userDefinedIdentifier* { PROG }

CODE → INSTR

CODE → INSTR ; CODE

# Grammar of an Instruction

---

INSTR  $\rightarrow$  halt

INSTR  $\rightarrow$  DECL

INSTR  $\rightarrow$  IO

INSTR  $\rightarrow$  CALL

INSTR  $\rightarrow$  ASSIGN

INSTR  $\rightarrow$  COND\_BRANCH

INSTR  $\rightarrow$  COND\_LOOP

# Grammar of IO

---

IO  $\rightarrow$  input ( VAR )

IO  $\rightarrow$  output ( VAR )

# Grammar of a Call

---

**CALL** → *userDefinedIdentifier*

# Grammar of a Declaration

---

DECL → TYPE NAME

DECL → TYPE NAME ; DECL



# Grammar of a Type

---

TYPE → num

TYPE → string

TYPE → bool

# Grammar of a Name

---

**NAME** → *userDefinedIdentifier*

# Grammar of a Variable

---

**VAR** → *userDefinedIdentifier*

# Grammar of an Assignment

---

**ASSIGN** → **VAR** = *stringLiteral*

**ASSIGN** → **VAR** = **VAR**

**ASSIGN** → **VAR** = **NUMEXPR**

**ASSIGN** → **VAR** = **BOOL**

# Grammar of a Numeric Expression

---

NUMEXPR → VAR

NUMEXPR → *integerLiteral*

NUMEXPR → CALC

# Grammar of a Calculation

---

**CALC** → add ( **NUMEXPR** , **NUMEXPR** )

**CALC** → sub ( **NUMEXPR** , **NUMEXPR** )

**CALC** → mult ( **NUMEXPR** , **NUMEXPR** )

# Grammar of if-statements

---

COND\_BRANCH  $\rightarrow$  if ( BOOL ) then { CODE }

COND\_BRANCH  $\rightarrow$  if ( BOOL ) then { CODE } else { CODE }

# Grammar of a Logic Expression

---

**BOOL** → eq ( **VAR** , **VAR** )

**BOOL** → ( **VAR** < **VAR** )

**BOOL** → ( **VAR** > **VAR** )

**BOOL** → not **BOOL**

**BOOL** → and ( **BOOL** , **BOOL** )

**BOOL** → or ( **BOOL** , **BOOL** )

**BOOL** → T

**BOOL** → F

**BOOL** → **VAR**



# Grammar of Loops

---

COND\_LOOP → while ( BOOL ) { CODE }

COND\_LOOP → for ( VAR = 0 ; VAR < VAR ; VAR = add ( VAR , 1 ) ) { CODE }

# SPL Program Example

---

```
input(x);  
n = x;  
r = "unknown";  
s = "even";  
checknumber;  
if (eq(r, s)) then {  
    output(s)  
} else {  
    output( r )  
};  
halt;
```

# Example (cont.)

---

```
proc checknumber {  
    num zero;  
    zero = 0;  
    m = n;  
    if (( m < zero ))  
        then { m = mult(m, -1) };  
    while ((m < zero)) { m = sub(m, 2) };  
    if (eq(m, zero))  
        then { r = "even" }  
        else { r = "odd" };  
}
```

# Your Tasks

---

- **Determine** with pen and paper **whether the SPL grammar is ambiguous.**
  - **If it is ambiguous: apply the appropriate grammar-transformation-techniques** to make it unambiguous. But **do not change the actual language SPL itself!**
    - **If you'd really want to work with an ambiguous SPL grammar, then you'd have to implement a GLR parser, which is difficult!**
- After you have dealt with the ambiguity question: **Determine** with pen and paper **whether you can write an LL(1) parser for your grammar:**
  - **If “yes”,** then write an LL(1) parser.
  - **If “no”,** then choose:
    - whether you want to write an SLR parser, or
    - whether you want to modify the grammar once more (without changing the SPL language itself!) in order to make the grammar suitable for LL(1) parsing.
- **Implement a parser** on the basis of chapter 2 of the book and on what you discovered in your theoretical analyses. **Help-Literature may be consulted.**

# Input

---

Your parser must be able to take in a text file that has an SPL program. Some files will be valid programs in SPL and some might be grammatically incorrect (i.e.: containing syntax errors).

**This input file should already be “tokenized” by your already existing lexer** before it gets consumed by your parser.

# Output

---

If your parser encounters something that it cannot parse it must output a descriptive error of where and why the error has occurred. For example the following could be the output that your parser prints:

**Syntax Error** [line: 2, col: 13]: Expected semicolon after INSTR.

**Syntax Error** [line: 4, col: 3]: Cannot compare EXPRs.

# Output (cont.)

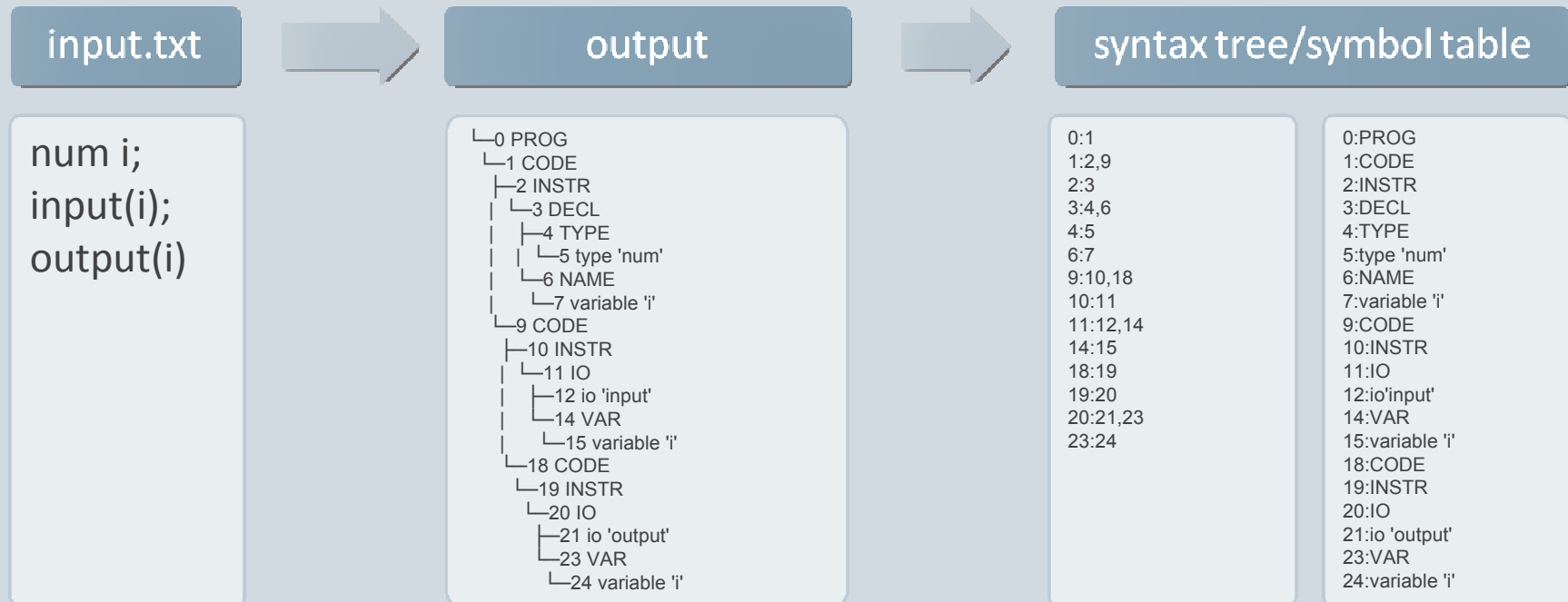
---

If no errors occur you must **first build a concrete syntax tree** which should **then be pruned creating an abstract syntax tree** (unnecessary tokens should be removed). **This abstract syntax tree and its symbol table must then be saved in (a) persistent file(s).**

- **Symbol Table → Forthcoming: Chapter 3** of our book.

**Each tree node should have a unique ID** and a pointer to its children nodes. **Each node should also have a pointer to the symbol table** where more information will be stored (**later**), such as each node's scope and value information, type information, etc...

# Output: Example



**Note:** The syntax tree represents its nodes in the form NodeID:ChildrenIDs...

**Note:** You do not have to structure your tree or table exactly as shown above nor do your ID's have to correspond to the above example.



# Additional Notes

---

- **You may not use any parser-generator tools!** Tools such as YACC and ANTLR are strictly forbidden. You must hand code your parser.
  - **Plagiarism is not allowed!** You or your group may not use any code written by someone not within your own group.
  - In the implementation language of your choice, **you may not use any libraries or built-in language features that provide parser-combinator-like features such as “fastparse”.**
- 

And now: HAPPY CODING 😊 😊 😊