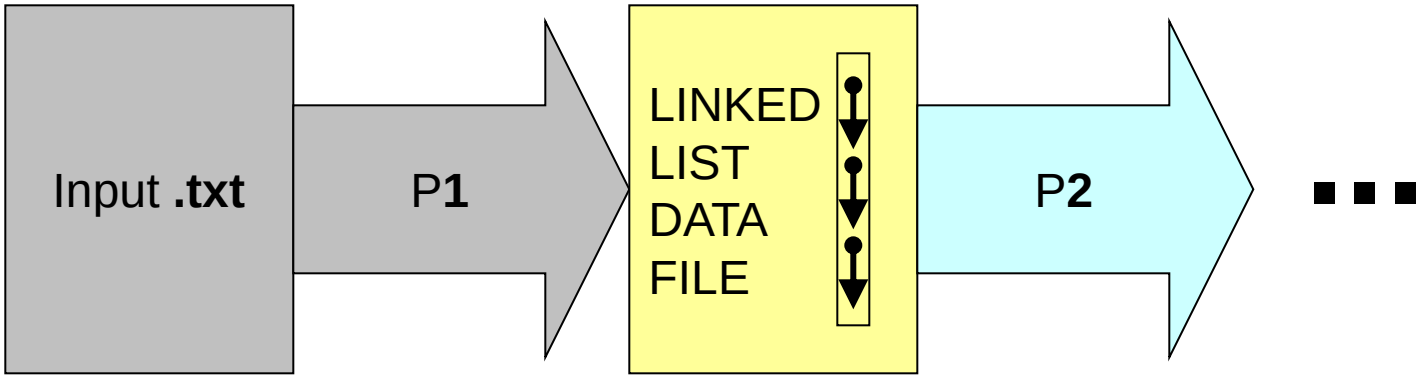
COS**341**, 2021

Practical**2**

SPECIFICATION

Preliminaries

- It is presumed that you have already successfully completed the previous Practical**1**
- The output file from P**1** will now serve as input file for P**2**.

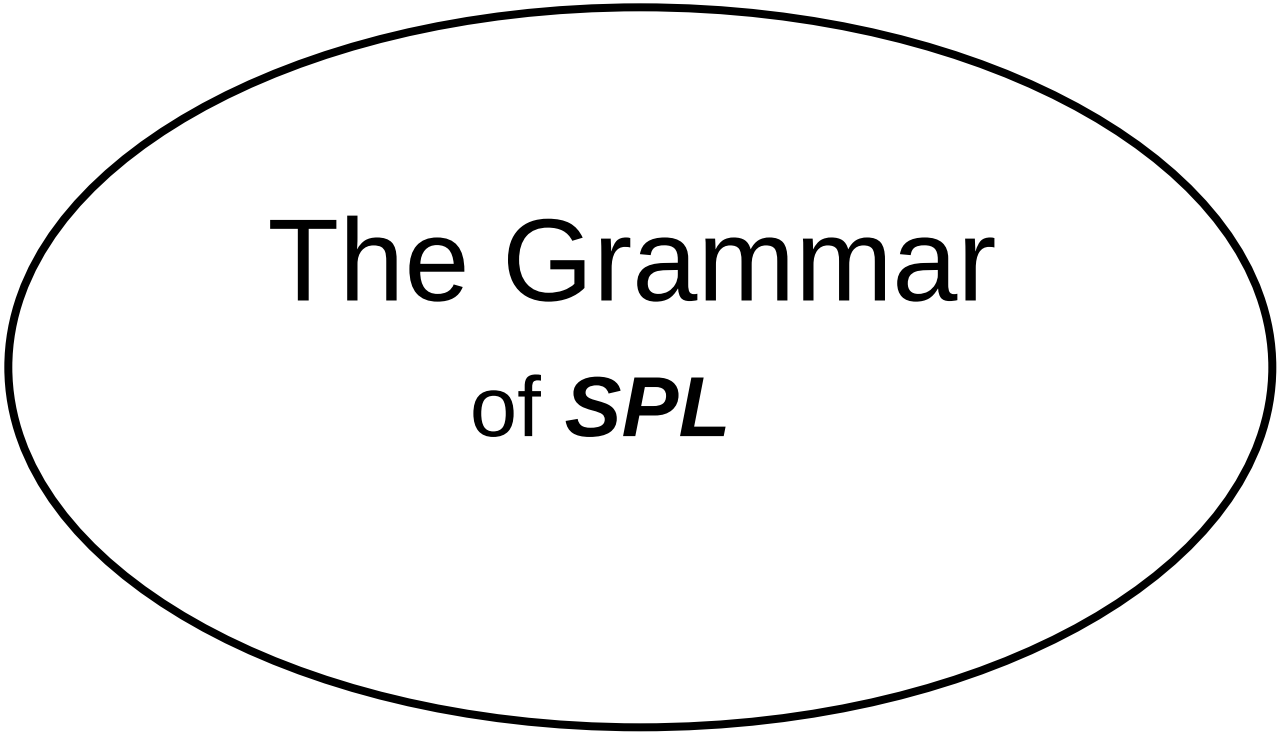


Students' Programming Language: ***SPL***

- Still we deal with ***SPL***, which professor has "designed" for you (for educational purposes).
  - In P1 we have already dealt with the *lexical analysis* of *SPL*.
    - **IF your Lexer did not work well on the Test-Day of P1, then please make sure that you fix all the identified Lexer-faults BEFORE you continue with this P2!**
      - *With a "buggy" Lexer, your Parser will not be able to correctly produce the required output!*

Students' Programming Language: ***SPL***

- On the following slides,
  - the ***syntactic structure*** of ***SPL*** will be given,
  - and your assignment task will be stipulated.


The Grammar
of ***SPL***

Notation

On the following slides,

- the RED CAPITAL LETTERS indicate the *Non*-terminal symbols in the grammar of ***SPL***
- the arrows ➔ separate the Left-hand-sides and the Right-hand-sides of the grammar's rules,
- the **blue bold** text indicates the *terminal tokens* in the *vocabulary* of ***SPL*** (from P1: "*lexing*"),
- normal text (like this) provides explanations and stipulates the project task which you will have to carry out.

- PROG → CODE
- PROG → CODE ; PROC_DEFS
- PROC_DEFS → PROC
- PROC_DEFS → PROC PROC_DEFS
- PROC → **proc** *UserDefinedName*
  { PROG **}**
- CODE → INSTR
- CODE → INSTR ; CODE

Note: the *UserDefinedName* comes from the **token class** of project P1 !

The bullet-points indicate the beginning of a new Rule of G;
The bullet-points do NOT belong to the grammar G itself.

- INSTR ➔ **halt**
- INSTR ➔ IO
- INSTR ➔ CALL
- INSTR ➔ ASSIGN
- INSTR ➔ COND_BRANCH
- INSTR ➔ COND_LOOP
- IO ➔ **input(**VAR**)**
- IO ➔ **output(**VAR**)**
- CALL ➔ *UserDefinedName*

Token-Class
identified by the Lexer

- VAR ➔ *UserDefinedName*

Token-Class
identified by the Lexer

- ASSIGN ➔ VAR = *String*
- ASSIGN ➔ VAR = VAR
- ASSIGN ➔ VAR = NUMEXPR
- NUMEXPR ➔ VAR
- NUMEXPR ➔ *Number*
- NUMEXPR ➔ CALC

Token-Class
identified by the Lexer

Token-Class
identified by the Lexer

- CALC → **add(**NUMEXPR**,**NUMEXPR**)**
- CALC → **sub(**NUMEXPR**,**NUMEXPR**)**
- CALC → **mult(**NUMEXPR**,**NUMEXPR**)**

- COND_BRANCH → **if(**BOOL**)**
  then{CODE**}**

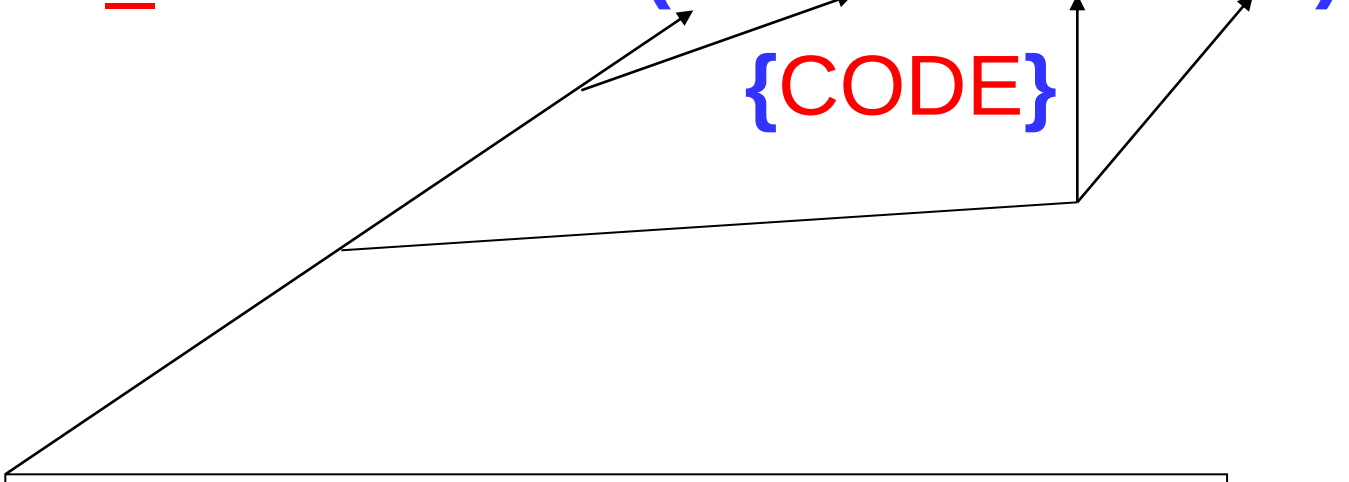
- COND_BRANCH → **if(**BOOL**)**
  then{CODE**}**
  else{CODE**}**

- BOOL → **eq(**VAR**,**VAR**)**
- BOOL → **eq(**BOOL**,**BOOL**)**
- BOOL → **eq(**NUMEXPR**,**NUMEXPR**)**
- BOOL → **(**VAR **<** VAR**)**
- BOOL → **(**VAR **>** VAR**)**
- BOOL → **not** BOOL
- BOOL → **and(**BOOL**,**BOOL**)**
- BOOL → **or(**BOOL**,**BOOL**)**

Note:*Later in another Practical,* our **Type Checker** shall deal with the question what to do if one of the variables is a string and the other one a number!

- COND_LOOP ➔ **while(**BOOL**)**
  {CODE**}**

- COND_LOOP ➔ **for(**VAR**=0** ; VAR**<**VAR ; VAR**=add(**VAR**,1)**)
  {CODE**}**



Note: *in a later Practical,* ***Static Semantic Analysis*** will have to guarantee that these 4 "VAR" are the same; otherwise the for-loop would not make any sense!
However, *in the syntax of a context-**free** grammar, we cannot capture any such context-dependencies!*

Some Additional Remarks

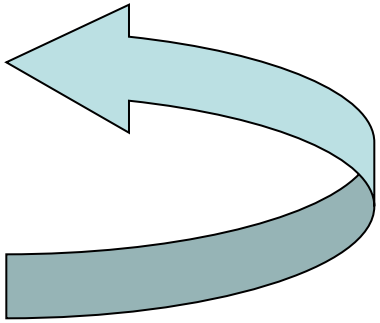
- The language ***SPL*** is *Turing-complete*: with it you can specify *all computable algorithms* in the domain of the Natural Numbers.
- On the following slides
  - you will see, for illustration, one small *example* of a meaningful ***SPL*** *program*
  - and you will get your TO-DO TASKS for this P**2**.

Small Example: **SPL Program**

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

continued on the next slide

}



Comment:
Procedure Calls
can have *Side Effects*
on the *Values* of *Variables*.
Later, in another Practical,
our **Scope Analyser**
will deal with the problem of
which Variables can be "seen"
from "where" in SPL programs,
(i.e.: "globally" or only "locally").

CODE inside checknumber{ ... }

```
m = n ;
if ( (m<0) )
   then { m = mult(m, –1) } ;
while( (m>0) )
   { m = sub(m,2) } ;
if( eq(m,0) )
   then { r = “even” }
   else { r = “odd” } ;
```

m is now a **local** variable which can be “*seen*” *only here inside* checknumber**!** The other variables are “seen” from “outside” of checknumber. **A later practical** will handle this scope-analysis.

YOUR TASKS

*for this P**2***

To do!

- **First, using the help of the book, analyse the given SPL grammar** with pen and paper:
  - Is it an "**ambiguous**" grammar**?**
    - **If yes**: can you remove some of the ambiguities by some suitable grammar-transformation-techniques without changing the language L(G)**?**
  - Is it suitable for **LL(1)** parsing on the basis of "FIRST", "FOLLOW", and "NULLABLE"**?**
    - **Hint**: Look particularly at **if(...)then{...}** versus **if(...)then{...}else{...}**
  - Is it suitable for **SLR** parsing without conflicts in the parse table**?**
    - are there any **shift-reduce** conflicts**?**
      - **If yes**: could you somehow "cope" with them**?**
    - are there any **reduce-reduce** conflicts**?**
      - **If yes**: could you somehow "cope" with them**?**

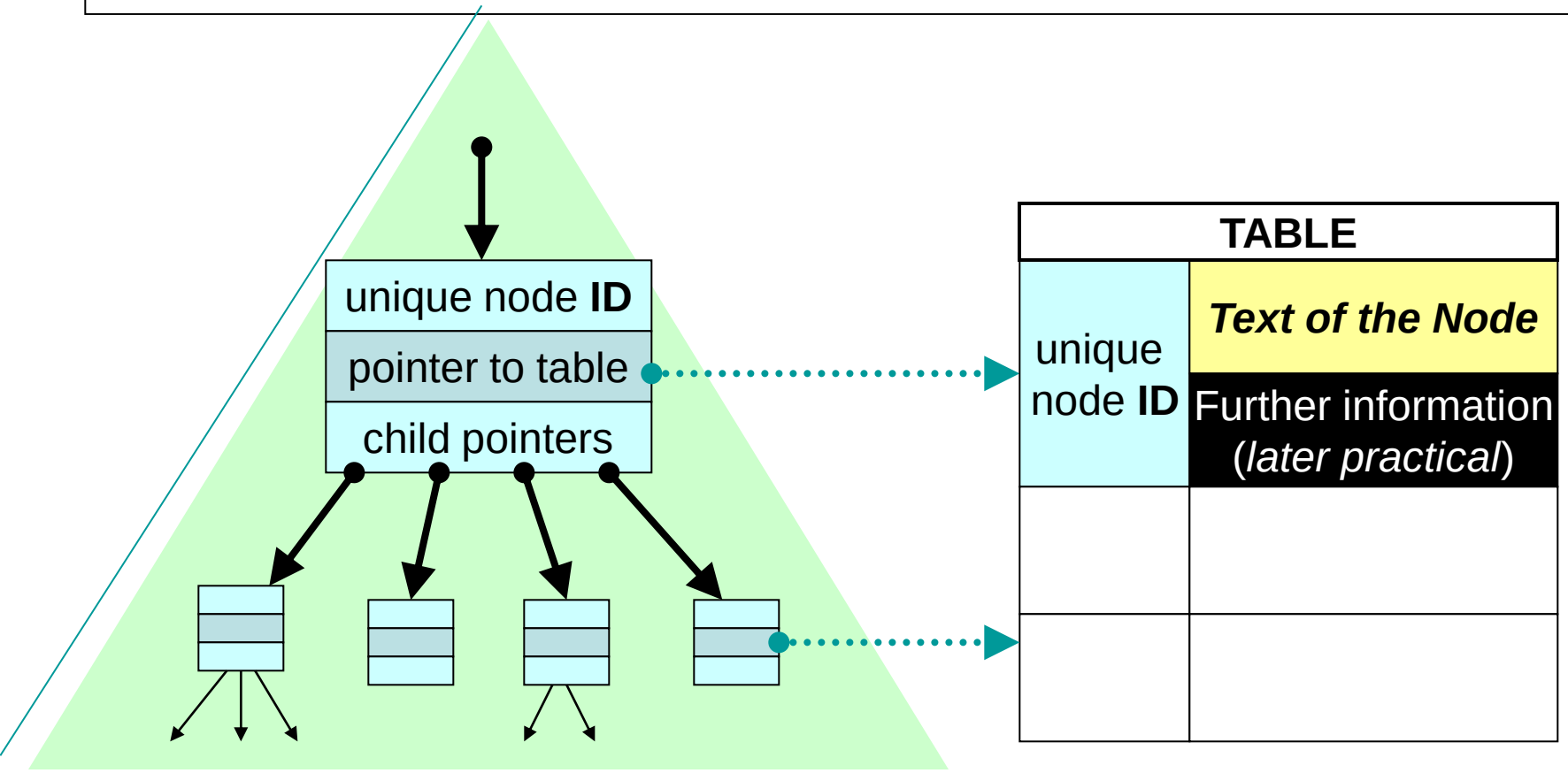
TO DO (Step-by-Step)

- On the basis of Chapter 2 of the book, **implement a Parser** that consumes the *token list from the input file* (P1) and attempts to create an **output file** that contains the corresponding **concrete syntax tree.**
  - **The parser must be hand-coded**!
    - no parser-generator tools (such as YACC or ANTLR) are allowed!
  - See the following slides for further details...
- On the basis of the distinction between **concrete** and **abstract** syntax (book Chapter 2), also implement a **Pruning software** which cuts all superfluous concrete symbols out of the file with the concrete syntax tree,
  - such that **only the** smaller **abstract syntax tree remains** standing **in the output file** of the Parser.

Additional Remarks on the Parser

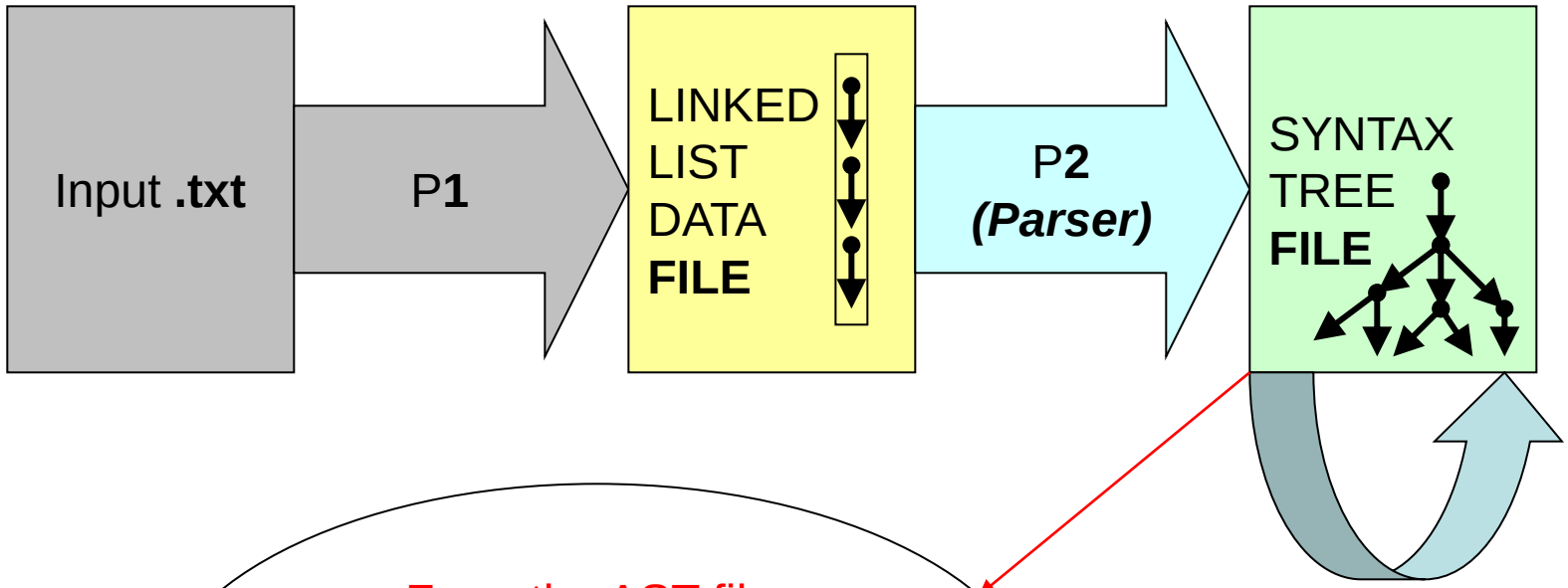
- If the parser encounters a **syntax error** in the grammatical structure of the string of tokens, the parser must output an error message which indicates to the user **where the error has happened and what kind of error it was**.
  - and then abort the parsing process (without continuing the Syntax Tree construction)
- The **Nodes in the Syntax Tree** are composite Data which must carry unique Node ID numbers as well as an additional pointer to an information table
  - which the compiler will later need for type checking and for scope analysis (*forthcoming practicals*)

Node in the *SPL* Syntax Tree



Text of the Node is either a NONTERMINAL from the Grammar, or a ***Terminal Token*** *identified by the Lexer.*

ALL IN ALL:


Input **.txt**
P**1**
LINKED
LIST
DATA
FILE
P**2**
(Parser)
SYNTAX
TREE
FILE
From the AST file,
a “visual representation” must be
transmitted to the client,
for intuitive “viewing”
(similar to a Directory Tree)
P**2**
(***Pruning***
concrete
to abstract
syntax tree)

Test Day =
Tuesday the
20th of April
2021

And **now**…

HAPPY CODING!

Note: Plagiarism is *forbidden!*
Code sharing with other students
is also *not allowed*