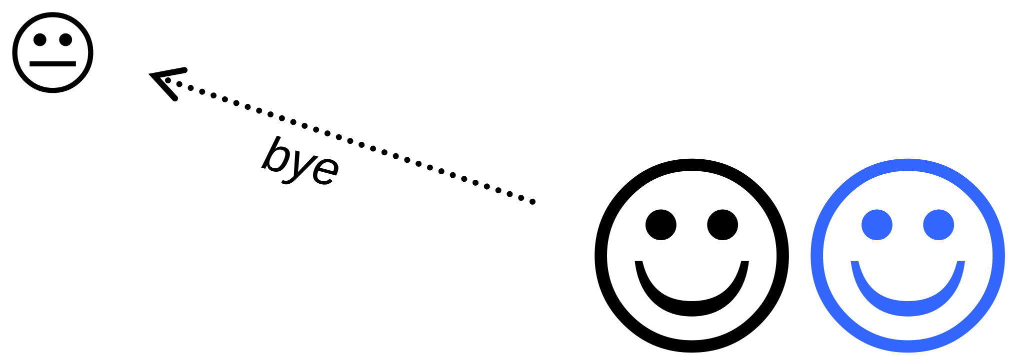
COS**341** Project **2** (2017):
Static Semantics of SPL

Part ***a***



NEW Project Partner !

will be allocated to you

during the lecture (soon)

Preliminaries

- **Two LEXICAL OBJECTS** (*Tokens* from the Lexer, *Leaf-Nodes* in the Parse-Tree) with the **same name X** refer to the same **VALUE OBJECT** (Memory Cell in RAM), **if** the following conditions hold:
  - The two X have the **same TYPE**, and
  - The two X belong to the **same SCOPE**.

Otherwise the two lexical objects refer to **different value objects** (in RAM) **even though both of them** happen to **carry the same name, X**

Preliminaries

- Target Code for two Lexical Objects with the same name X can thus be generated correctly only when the Compiler “knows”
  - whether the two same-name-X *are meant to represent* the same value object (in RAM) or
  - whether the two same-name-X are *meant to represent* different value objects (in RAM).
- This kind of *meaning* is captured by the SPL’s **Static Semantics**.

Two kinds of **SEMANTICS**

- **DYNAMIC Semantics**
  - *Describes formally* ***what happens*** *when a Program* ***is running****.*
- **STATIC Semantics**
  - *Describes formally the* ***Types*** *and* ***Scope-Relations among the Lexical Objects*** *which occur in a Program (before it runs).*
    - **Note that these semantic relations are NOT already captured in the Syntax Grammar of the Programming Language!**

Consequently:

- After the Parser has generated the Syntax Tree, *Static Semantic Analysis* (for Types and Scopes) is a *necessary* phase of the Compiler before any Target Code can be correctly generated.
- **All this is the topic of this Project2**!
  - Part 2***a*** (***this*** part) deals with the **Type Checking**
    - Note: If two same-named lexical objects X *have different Types*, then they *cannot refer to the same value object* in RAM, *regardless of the scope(s)* in which they may stand!
  - Part 2***b*** (*later*) will deal with the **Scope Analysis**
    - *Only needed for* two same-name-X *of the same type*; thanks to the foregoing Type Analysis!

Consequently:

- After the Parser has generated the Syntax Tree, *Static Semantic Analysis* (for Types and Scopes) is a *necessary* phase of the Compiler before any Target Code can be correctly generated.
- **All this is the topic of this Project2**!
  - Part 2***a*** (***this*** part) deals with the **Type Checking**
  - Part 2***b*** (*later*) will deal with the **Scope Analysis**
    - ***Scope Analysis will then, eventually, also enable us****:*
      - *to find out, within each scope, whether every used variable has received a definite value before its further usage, and*
      - *to give to each value-object (in RAM) a unique new variable name, i.e.: to remove the confusing same-naming of different value-objects*

Example Scenario (***SPL***)

x = **1** ;

x = "***john***" ;

x = **add**(x,"***lisa***") ;

output(x)

STOP !
Look at this Slide
for at least one minute
before you switch to
the following Slide

Example Scenario (***SPL***)

x = **1** ;

x = "***john***" ;

These **two x represent different objects in RAM**:
The one is a Number_x,
The other one is a String_x

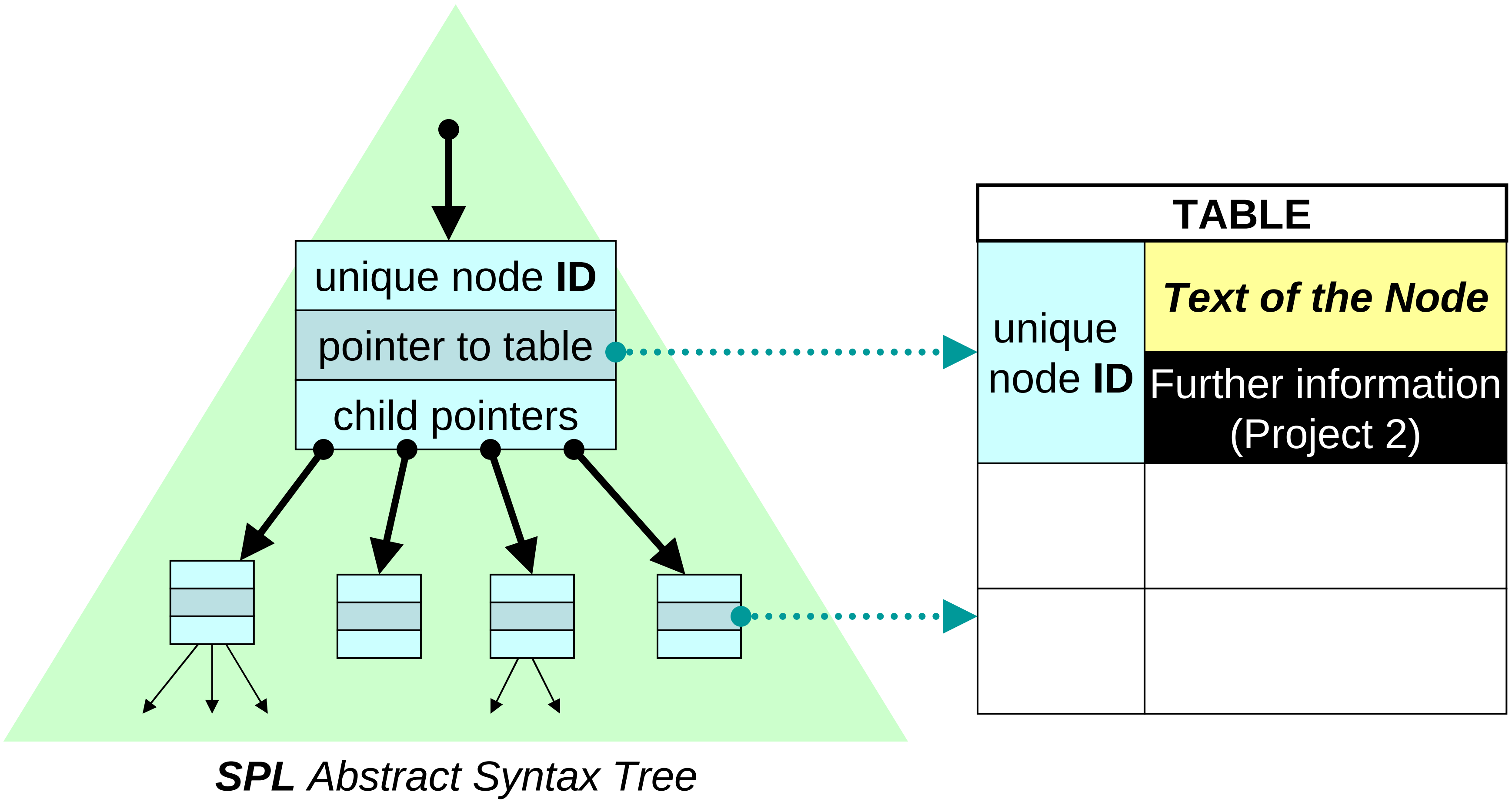
x = **<u>add</u>**(x,<u>"***lisa***"</u>) ;

Type Error :
The **add** operation is **not allowed for Strings !**

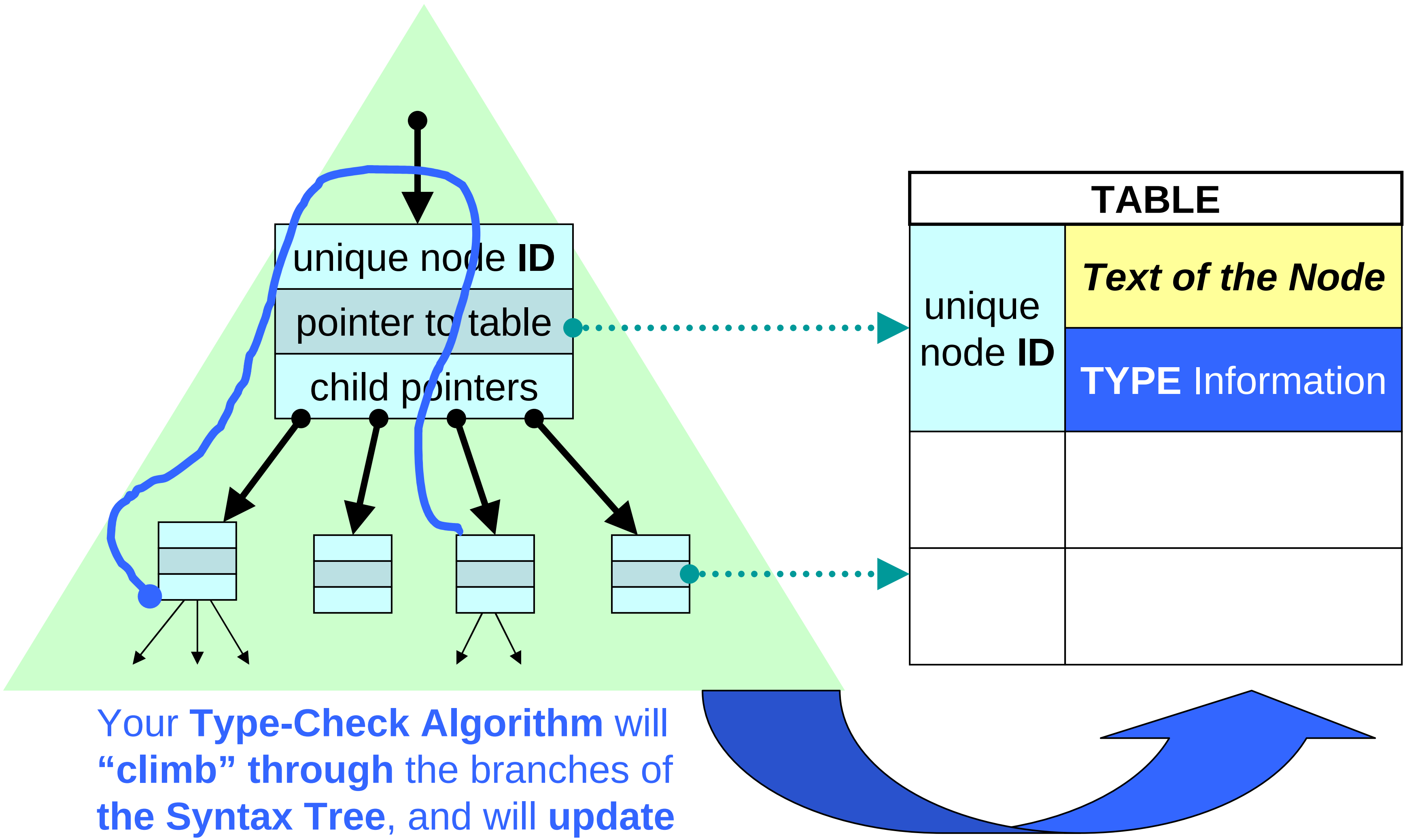
output(x)

Which one
of the two different x
shall be the output **??**

Remember from **Project 1**:



Now in **Project 2*a***:



Your **Type-Check Algorithm** will **"climb" through** the branches of **the Syntax Tree**, and will **update the Information Table** accordingly!

However:

- Before we can implement the tree-climbing Type-check Algorithm,

  *we need to know its **LOGICAL RULES**.*
  - **The Type-Checker's Logical Rules "tell" the Algorithm**:
    - Under which circumstances to **output "ERROR"**, respectively
    - Under which circumstances to **accept the** given **Syntax-Tree** as **correctly typed**.

End of "Preliminaries"

The Grammar-Based **TYPE RULES**

of *SPL*

(Please recall the Grammar of *SPL* from Project 1)

INITIALISATION:

- **At the very beginning**, for ***each node N*** *in the Syntax Tree*:
  - **set its type information** [in the table] to: ‘’ (the empty character)
    - Note: we will **use** single-**characters** (*not* strings) **to represent types** in our Information Table.
- Thereafter, the type-checking algorithm can climb up/down through the branches of the Tree, and **update** type information according to the following **rules**:

- PROG ➔ CODE

if PROG**.*type*** == **‘’**
and if CODE**.*type*** == **‘w’** *// well-typed*
then update PROG**.*type*** := **‘w’**

Implementation **Hint**:
At the very end of all Type-Checking, **NO** node **IN** the Tree is allowed to be affiliated **with the initial type** symbol **‘’** any longer ➔ *otherwise:* **Type Error !**

- PROG → CODE ; PROC_DEFS

if PROG.***type*** **== ‘’**

and if CODE.***type*** **== ‘w’**

and if PROC_DEFS.***type*** **== ‘w’**

then update PROG.***type*** **:= ‘w’**

- PROC_DEFS → PROC
- PROC_DEFS → PROC PROC_DEFS

Similar type rules like in the previous two examples

- PROC → **proc** *UserDefinedName* **{** PROG **}**

```
if ( PROC.type == UserDefinedName.type == ‘’ )
and if PROG.type == ‘w’
then {
        update UserDefinedName.type := ‘p’ // procedure
        update PROC.type := ‘w’
    }
```

- CODE → INSTR
- CODE → INSTR **;** CODE

Similar type rules like on the previous slides

- INSTR → **halt**

if INSTR.***type*** == **‘’**
then update INSTR.***type*** := **‘w’**

- INSTR → IO

if INSTR.***type*** == **‘’**
and if IO.***type*** == **‘w’**
then update INSTR.***type*** := **‘w’**

- INSTR → CALL

Similar type rule as above

- INSTR → ASSIGN
- INSTR → COND_BRANCH
- INSTR → COND_LOOP

Similar Type Rules as on the previous slides

- IO → **input(VAR)**

```
// For the sake of simplicity we only
// allow the user to interactively enter
// numbers in our education language SPL
if IO.type == ‘’
then { update VAR.type := ‘n’ // number
       update IO.type := ‘w’ }
```

- IO → **output(VAR)**

```
// In SPL, our interactive screen-display
// shall only show strings or numbers
if IO.type == ‘’
then { update VAR.type := ‘o’ // output
       update IO.type := ‘w’ }
         // Soon we will see that the o-type can
         // serve for strings as well as for numbers
```

- CALL → *UserDefinedName*

```
if CALL.type == ‘’
then {
       update UserDefinedName.type := ‘p’  // procedure type
       update CALL.type := ‘w’                                }
```

- VAR → SVAR
- VAR → NVAR
- SVAR → *UserDefinedName*
- NVAR → *UserDefinedName*

HINT:
In the original ***SPL*** Grammar
there is some ambiguity at this place,
because we cannot see from a ***UserDefinedName***
whether a **VAR** is meant to be an **SVAR** or an **NVAR**.
If you have modified the original ***SPL*** Grammar earlier in **Project 1*a***,
such as to get rid of this grammatical ambiguity,
then you might have to modify the following Typing Rules accordingly.

- VAR → SVAR

if VAR***.type*** **== 'o'**
then update SVAR***.type*** **:= 's'** *// **s**tring*

- VAR → NVAR

(if VAR***.type*** **== 'o'**
or if VAR***.type*** **== 'n')**
then update NVAR***.type*** **:= 'n'** *// **n**umber*

- SVAR → *UserDefinedName*

if SVAR***.type*** **== 's'**
then update *UserDefinedName****.type*** **:= 's'**

- NVAR → *UserDefinedName*

if NVAR***.type*** **== 'n'**
then update *UserDefinedName****.type*** **:= 'n'**

- ASSIGN → SVAR = *String*
- ASSIGN → SVAR = SVAR
- ASSIGN → NVAR = NUMEXPR
- NUMEXPR → NVAR
- NUMEXPR → *Number*
- NUMEXPR → CALC

HINT:
Also at this point there is some ambiguity
in the original grammar of ***SPL***;
for example:
we cannot guess from the pure variable names if **x = y**
is an ASSIGN with SVAR and SVAR,
or an ASSIGN with NVAR and NVAR.
Again it is presumed that you have "intelligently dealt with"
such ambiguities in the previous Project **1*b*.**

- ASSIGN → SVAR = *String*

```
if ( ASSIGN.type == '' )
then { update String.type := 's'
       update SVAR.type := 's'
       update ASSIGN.type := 'w' }
```

- ASSIGN → SVAR[1st] = SVAR[2nd]

```
if ( ASSIGN.type == '' )
then { update SVAR1st .type := 's'
       update SVAR2nd .type := 's'
       update ASSIGN.type := 'w' }
```

- ASSIGN → NVAR = NUMEXPR

```
if ( ASSIGN.type == '' )
and if NUMEXPR.type == 'n'
then { update NVAR.type := 'n'
       update ASSIGN.type := 'w' }
```

- NUMEXPR → NVAR

if NUMEXPR.***type*** == ‘’
then { **update** NVAR.***type*** := ‘**n**’
update NUMEXPR.***type*** := ‘**n**’ }

- NUMEXPR → *Number*

if NUMEXPR.***type*** == ‘’
then { **update** *Number*.***type*** := ‘**n**’
update NUMEXPR.***type*** := ‘**n**’ }

- NUMEXPR → CALC

if NUMEXPR.***type*** == ‘’
and if CALC.***type*** == ‘**n**’
then update NUMEXPR.***type*** := ‘**n**’

- CALC → **add(**NUMEXPR^{1st}**,**NUMEXPR^{2nd}**)**

> **if** CALC***.type*** == **‘’**
> **and if** (NUMEXPR^{1st}***.type*** == NUMEXPR^{2nd}***.type*** == ‘**n**’)
> **then update** CALC***.type*** := ‘**n**’

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

> Same type rules as in the example of above

- COND_BRANCH → **if(**BOOL**)**

  then{ CODE **}**

```
if COND_BRANCH.type == ‘’
and if BOOL.type == ‘b’ // boolean
and if CODE.type == ‘w’
then update COND_BRANCH.type := ‘w’
```

- COND_BRANCH → **if(**BOOL**)**

  then{ CODE^{1st} **}**

  else{ CODE^{2nd} **}**

Typing rule similar as above

- BOOL → **eq(**VAR^{1st}**,**VAR^{2nd} **)**

```
if BOOL.type == ‘’
// Remark: here we are allowed to compare anything with anything
then { update BOOL.type := ‘b’
       update VAR1st.type := ‘o’
       update VAR2nd.type := ‘o’ }
```

- BOOL → **(**NVAR^{1st} **<** NVAR^{2nd} **)**

```
if BOOL.type == ‘’
then { update BOOL.type := ‘b’
       update NVAR1st.type := ‘n’
       update NVAR2nd.type := ‘n’ }
```

- BOOL → **(**NVAR **>** NVAR **)**

Type Rule as above

- BOOL → **not** BOOL
- BOOL → **and(**BOOL**,**BOOL**)**
- BOOL → **or(**BOOL**,**BOOL**)**

- COND_LOOP → **while(**BOOL**)**

  {CODE**}**

Type Rule very similar as for COND_BRANCH

- COND_LOOP → **for(**NVAR**=0 ;** NVAR**<**NVAR **;** NVAR**=add(**NVAR**,1)** **)**

  {CODE**}**

EXTREMELY IMPORTANT **!!!**

- The *typing rules on the previous slide* are given for the *unmodified original grammar* of *SPL*.
- **However**: for the Parser in **Project 1**, you had been asked to *modify the grammar* (*if necessary*), such as to make the grammar parseable.
- **Thus**: ***if** you had modified the grammar* in Project 1, *you must accordingly modify* the *Typing Rules* (on the previous slides), *too*!

Thus you have got an impression **how type checkers work in general**:

To each production rule of a programming language's grammar, there is a corresponding type checking rule.
After the parser has successfully constructed a syntax tree, the type checker "visits" all the nodes of the tree.
At each node "visited" the type checker attempts at applying the type-checking rule which is affiliated with its corresponding grammar rule at that "place" in the tree.

Thereby, two different kinds of type checkers can be distinguished:

In programming languages, in which the users themselves must declare all types to their variables, the type checker assesses if the user has done this job correctly; (example: JAVA)
In programming languages like our *SPL*, in which the *user is **not** asked to declare 'upfront' the types of the variables*, *the checker* 'tries to make sense' of the user's program, and *tries to allocate automatically the right types* to the user's un-declared variables. **Type checkers of this kind are also called type inferencers**.

Your TASK:

IMPLEMENT and TEST

an *SPL* **Type Checker**

And **now**…

HAPPY PAIR-**CODING**!

☺☺

Note: Plagiarism is forbidden! Code swapping with other pairs of project students is also not allowed