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

Part ***c***

VALUE ANALYSIS

Phase *A*

Unique Re-Naming

In sub-projects 2***a*** and 2***b*** we had already learned...

- If *two same-named lexical objects* (tokens) have the same type and stand in the same scope,
  - then they also refer to the same value object (in RAM)
- If *two same-named lexical objects* (tokens) are of different types or stand in separated scopes
  - then they refer to different value objects (in RAM), in spite of their same names.

Consequently...

- We can now *consistently* ***re-name*** all *UserDefinedNames* in the Abstract Syntax Tree by system-generated unique names, such that ***no more same-naming for different value-objects*** (in RAM) occurs.

  – This automatic renaming **will make the** subsequent **value analysis** considerably **easier**!

EXAMPLE

with user-names

```
x = 12 // number
z = 9 // number
x = “hello” // string
x // procedure call
output(x)
proc x
   {
    z = add(x,z)
   }
```

after re-naming

```
n0 = 12 // number
n1 = 9 // number
s0 = “hello” // string
p0 // procedure call
output(s0)
proc p0
   {
    n1 = add(n0,n1)
   }
```

All name-ambiguities have vanished

Sub-Task for Phase *A*

- Implement an algorithm which “crawls through” the Abstract Syntax Tree, and which *correctly and consistently re-names all UserDefinedNames* such that **no more same-name-ambiguities remain**.
- Thereby exploit the already computed type information (from **Project 2*a***) as follows:
  - For **numeric** variables: **n**0, **n**1, **n**2, **n**3, ...
  - For **string** variables: **s**0, **s**1, **s**2, **s**3, ...
  - For **procedure** names: **p**0, **p**1, **p**2, **p**3, ...

Phase *B*

Appl-Decl-Paths

Look at the following **re-named** ***SPL*** Program:

```
input(n0)
n1 = add(n0,n2)
output(n2)
input(n2)
```

Look at the following **re-named** ***SPL*** Program:

```
input(n0)
n1 = add(n0,n2)
output(n2)
input(n2)
```

SYNTACTICALLY
it is CORRECT !
There is nothing wrong with its Syntax Tree

Look at the following **re-named** ***SPL*** Program:

```
input(n0)
n1 = add(n0,n2)
output(n2)
input(n2)
```

however :
its **Static Semantics**
is FLAWED !

Look at the following **re-named** ***SPL*** Program:

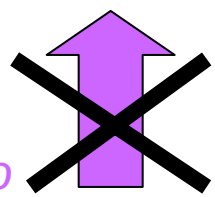
Only here it is right: ***n0*** *has* ***received a value*** *that can be subsequently used*

```
input(n0)
n1 = add(n0,n2)
output(n2)
input(n2)
```

Variable ***n2*** *has* ***no*** *value at this point* ***!***

Hence also ***n1*** *has* ***no*** *value here* ***!***

The input at this late point ***is useless****, because the* ***value cannot flow*** *“****upwards****” to where it would be needed*



however :
its **Static Semantics is FLAWED !**

Look at the following **re-named** ***SPL*** Program:

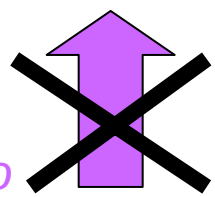
Only here it is right: ***n0*** *has* ***received a value*** *that can be subsequently used*

```
input(n0)
n1 = add(n0,n2)
output(n2)
input(n2)
```

Variable ***n2*** *has* ***no*** *value at this point* ***!***

Hence also ***n1*** *has* ***no*** *value here* ***!***

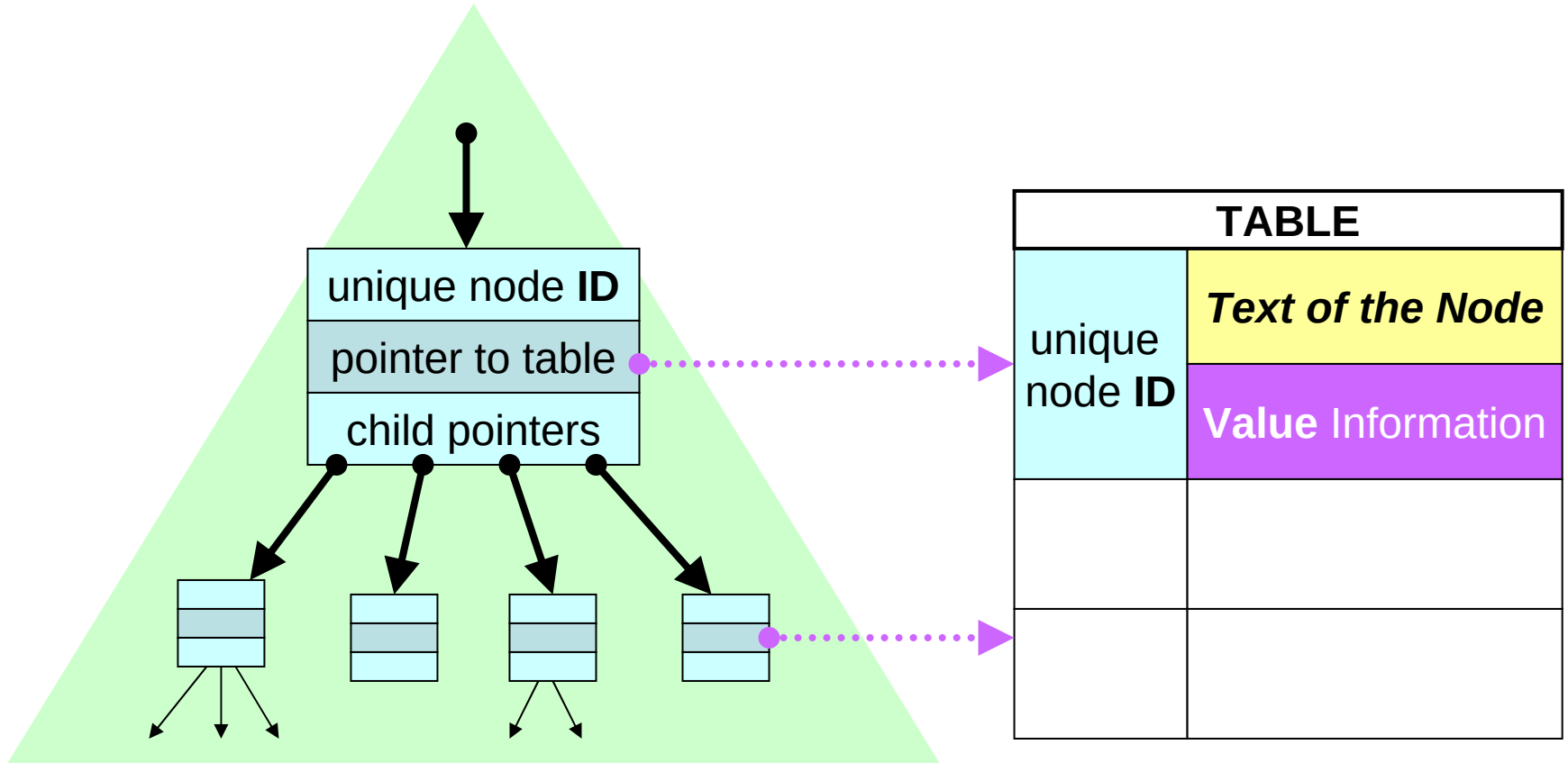
The input at this late point ***is useless****, because the* ***value cannot flow*** *"****upwards****" to where it would be needed*



however :
its **Static Semantics is FLAWED !**

Hence, our compiler **MUST NOT create target code** for this SPL program**!**

Node in the *SPL* Syntax Tree



Thus, in this Phase *B* of **Sub-Project 2*c*** we must now "crawl through" the Syntax Tree once more, and **record for each node with a name** (n0, n1, ..., s0, s1, ... p0, p1...) **whether or not it has a usable value**
➔ **This is also known as Application-Declaration-Path-Analysis.**

Initialisation

- Initially, for each node in the Syntax Tree the corresponding **value indicator in the Table** is the character '**u**' (for "unknown")

Steps of the Algorithm

- Step by step, the analysis algorithm tries to replace the '**u**' information by the '**d**' information ("defined") wherever a name appears in an "***applied*** *occurrence*" (i.e.: on the right-hand-side of an assignment instruction) in the Syntax Tree.
- Such replacement is **easy**, if we can find the same name (thanks to our re-naming!) in a "*defined occurrence*" at a "correlated" position in the Syntax Tree.

Termination

- The algorithm emits "**ERROR**" if there exists in the Syntax Tree a **name** in an "*applied*" position for which the initial '**u**' information **cannot be replaced by** '**d**'.

  – *In such error case, the compiler will not be allowed to generate machine code for this Syntax Tree.*

Value Flow Rules

on which the Analysis Algorithm is based

- "**Applied**" Positions are:
  - The variable in an **output(**VAR**)** instruction
  - The **p**-name in an SPL *procedure call* **p ;**
  - All variables in the RIGHT-HAND-SIDE of Assignment instructions

    VAR **=** (RIGHT-HAND-SIDE)

Implementation Hint:
You might perhaps want to use the character '**a**' in the **information table** to indicate if some AST node occurs in an "applied" position.

Value Flow Rules

on which the Analysis Algorithm is based

- "**Defining**" Positions are:
  - The variable in an **input(**VAR**)** instruction
  - The **p**-name in a *declaration* **proc p { ... }**
  - The variable on the LEFT-HAND-SIDE of Assignment instructions

    VAR = (RIGHT-HAND-SIDE)

Value Flow Rules

on which the Analysis Algorithm is based

- '**u**'-'**d**'-Replacement for **input(**VAR**)**
  - Any variable in this situation is *interactively instantiated by the user*; its '**u**'-label can be replaced immediately by '**d**'.
- '**u**'-'**d**'-Replacement for constant **strings**:
  - Any *constant string node* (e.g.: **"hello"**) in the AST *has a defined value* and gets its '**u**'-label replaced by a '**d**' label immediately.
- '**u**'-'**d**'-Replacement for constant **numbers**:
  - Ditto for constant number nodes (e.g.: **–17**)

Value Flow Rules

on which the Analysis Algorithm is based

- '**u**'-'**d**'-Replacement for Assignments (**1**):
  - The '**u**' label on the LEFT-HAND-SIDE of an Assignment instruction is replaced by the '**d**' label **if all** AST nodes in the Assignment's RIGHT-HAND-SIDE are already '**d**'-labeled.

Value Flow Rules

on which the Analysis Algorithm is based

- '**u**'-'**d**'-Replacement for Assignments (**2**):
  - An '**u**' label in the RIGHT-HAND-SIDE of an Assignment instruction is replaced by the '**d**' label
    - **if** the corresponding *applied* variable of name *X* finds its *defined* "partner" of the same name *X* at an "earlier position" in the AST,
    - **and if** that corresponding defined "partner" already carries the label '**d**' itself.

Ditto for the situation: **output(**VAR**)**

Value Flow Rules

on which the Analysis Algorithm is based

- '**u**'-'**d**'-Replacement for: **proc P { CODE }**
  - The '**u**' label on the **P**-name of an *procedure declaration* is replaced by the '**d**' label **if all** AST nodes inside the procedure's CODE are already '**d**'-labeled.

Value Flow Rules

on which the Analysis Algorithm is based

- '**u**'-'**d**'-Replacement for *procedure call* **P ;**
  - The '**u**' label on the **P**-name of an *procedure call* is replaced by the '**d**' label, **if** at a "*later* place" in the AST a corresponding procedure *definition* (with same name **P**) is found which already carries the '**d**' label itself.

Value Flow Rules

on which the Analysis Algorithm is based

- Similar rules can be easily stipulated also for all the other constructs of *SPL*, such as the WHILE-statements with their Boolean parameters, the IF-THEN-ELSE-branches with their Boolean parameters, etc...

TO DO: Students !

An **Example Run** of the Analyser

n0 = *12*

input(**n1**)

n2 = *0*

p0

output(**n2**)

proc **p0** { input(**n3**)

n4 = add(**n3**,**n0**)

n2 = mul(**n1**,**n4**) }

An **Example Run** of the Analyser

n0 = *12*

input(**n1**)

n2 = *0*

p0

output(**n2**)

proc **p0** { input(**n3**)

n4 = add(**n3**,**n0**)

n2 = mul(**n1**,**n4**) }

An **Example Run** of the Analyser

n0 = *12*

input(**n1**)

n2 = *0*

p0

output(**n2**)

proc **p0** { input(**n3**)

n4 = add(**n3**,**n0**)

n2 = mul(**n1**,**n4**) }

An **Example Run** of the Analyser

n0 = *12*

input(**n1**)

n2 = *0*

p0

output(**n2**)

proc **p0** { input(**n3**)

n4 = add(**n3**,**n0**)

n2 = mul(**n1**,**n4**) }

An **Example Run** of the Analyser

n0 = *12*

input(**n1**)

n2 = *0*

p0

output(**n2**)

proc **p0** { input(**n3**)

n4 = add(**n3**,**n0**)

n2 = mul(**n1**,**n4**) }

An **Example Run** of the Analyser

n0 = *12*

input(**n1**)

n2 = *0*

p0

output(**n2**)

proc **p0** { input(**n3**)

n4 = add(**n3**,**n0**)

n2 = mul(**n1**,**n4**) }

An **Example Run** of the Analyser

n0 = *12*

input(**n1**)

n2 = *0*

p0

output(**n2**)

proc **p0** { input(**n3**)

n4 = add(**n3**,**n0**)

n2 = mul(**n1**,**n4**) }

An **Example Run** of the Analyser

n0 = *12*

input(**n1**)

n2 = *0*

p0

output(**n2**)

proc **p0** { input(**n3**)

n4 = add(**n3**,**n0**)

n2 = mul(**n1**,**n4**) }

An **Example Run** of the Analyser

n0 = *12*

input(**n1**)

n2 = *0*

p0

output(**n2**)

proc **p0** { input(**n3**)

n4 = add(**n3**,**n0**)

n2 = mul(**n1**,**n4**) }

An **Example Run** of the Analyser

n0 = *12*

input(**n1**)

n2 = *0*

p0

output(**n2**)

proc **p0** { input(**n3**)

n4 = add(**n3**,**n0**)

n2 = mul(**n1**,**n4**) }

An **Example Run** of the Analyser

```
n0 = 12
input(n1)
n2 = 0
p0
output(n2)
proc p0 {  input(n3)
           n4 = add(n3,n0)
           n2 = mul(n1,n4)  }
```

An **Example Run** of the Analyser

```
n0 = 12
input(n1)
n2 = 0
p0
output(n2)
proc p0 {  input(n3)
           n4 = add(n3,n0)
           n2 = mul(n1,n4)  }
```

An **Example Run** of the Analyser

```
n0 = 12
input(n1)
n2 = 0
p0
output(n2)
proc p0 {  input(n3)
           n4 = add(n3,n0)
           n2 = mul(n1,n4)  }
```

An **Example Run** of the Analyser

n0 = *12*

input(**n1**)

n2 = *0*

p0

output(**n2**)

proc **p0** { input(**n3**)

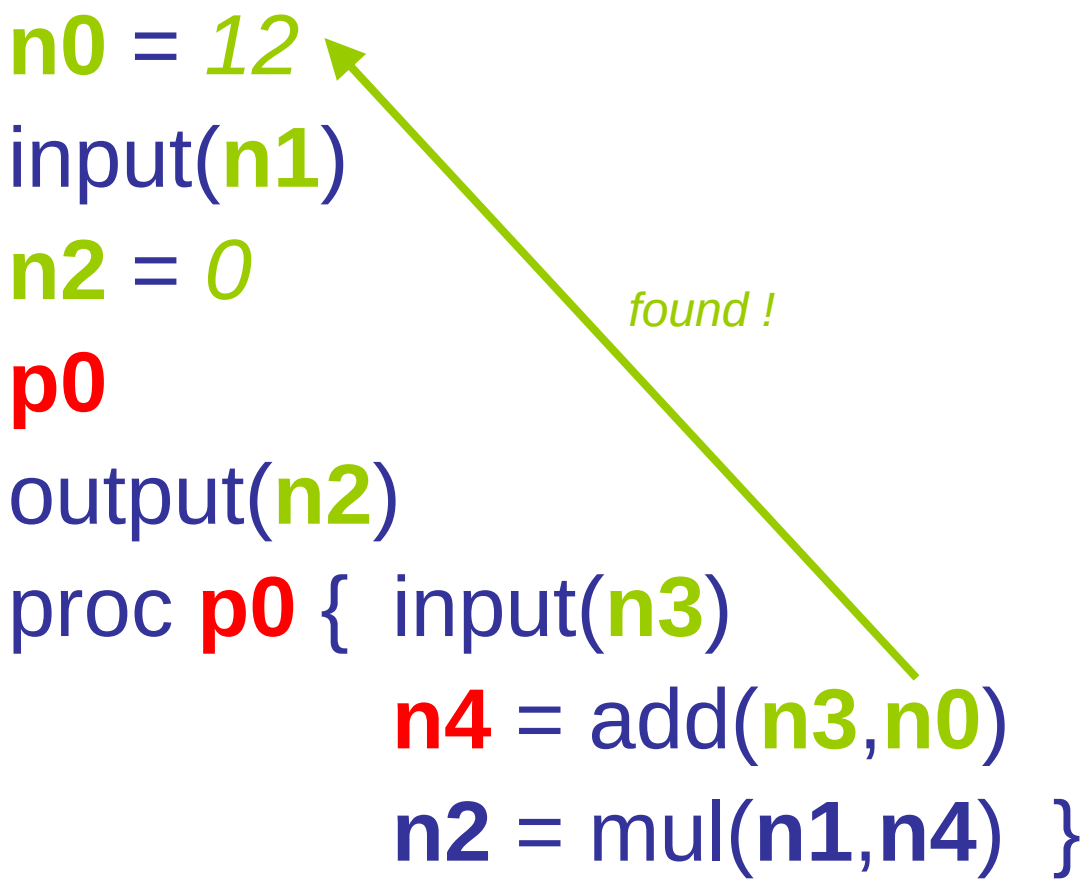
n4 = add(**n3**,**n0**)

n2 = mul(**n1**,**n4**) }

An **Example Run** of the Analyser

```
n0 = 12
input(n1)
n2 = 0
p0
output(n2)
proc p0 {  input(n3)
           n4 = add(n3,n0)
           n2 = mul(n1,n4)  }
```

An **Example Run** of the Analyser



An **Example Run** of the Analyser

```
n0 = 12
input(n1)
n2 = 0
p0
output(n2)
proc p0 {  input(n3)
           n4 = add(n3,n0)
           n2 = mul(n1,n4)  }
```

An **Example Run** of the Analyser

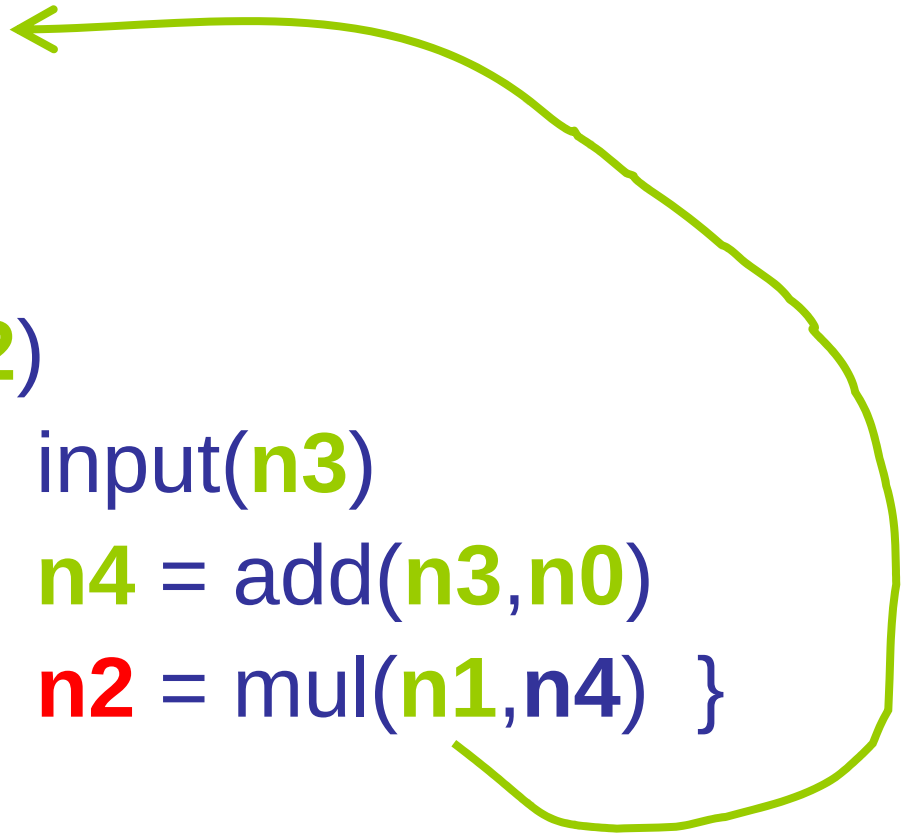
```
n0 = 12
input(n1)
n2 = 0
p0
output(n2)
proc p0 {  input(n3)
           n4 = add(n3,n0)
           n2 = mul(n1,n4)  }
```

An **Example Run** of the Analyser

```
n0 = 12
input(n1)
n2 = 0
p0
output(n2)
proc p0 {  input(n3)
           n4 = add(n3,n0)
           n2 = mul(n1,n4)  }
```

An **Example Run** of the Analyser

```
n0 = 12
input(n1)
n2 = 0
p0
output(n2)
proc p0 {  input(n3)
           n4 = add(n3,n0)
           n2 = mul(n1,n4)  }
```

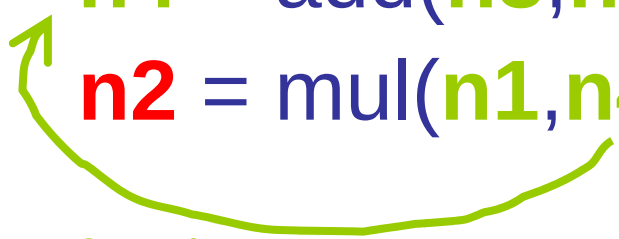


An **Example Run** of the Analyser

```
n0 = 12
input(n1)
n2 = 0
p0
output(n2)
proc p0 {  input(n3)
           n4 = add(n3,n0)
           n2 = mul(n1,n4)  }
```

An **Example Run** of the Analyser

```
n0 = 12
input(n1)
n2 = 0
p0
output(n2)
proc p0 {  input(n3)
           n4 = add(n3,n0)
           n2 = mul(n1,n4)  }
```



An **Example Run** of the Analyser

```
n0 = 12
input(n1)
n2 = 0
p0
output(n2)
proc p0 {  input(n3)
           n4 = add(n3,n0)
           n2 = mul(n1,n4)  }
```

An **Example Run** of the Analyser

```
n0 = 12
input(n1)
n2 = 0
p0
output(n2)
proc p0 {  input(n3)
           n4 = add(n3,n0)
           n2 = mul(n1,n4)  }
```

An **Example Run** of the Analyser

n0 = *12*

input(**n1**)

n2 = *0*

p0

output(**n2**)

proc **p0** { input(**n3**)

n4 = add(**n3**,**n0**)

n2 = mul(**n1**,**n4**) }

An **Example Run** of the Analyser

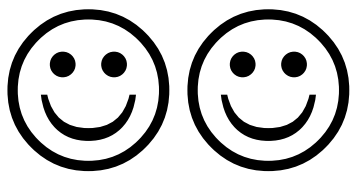
```
n0 = 12
input(n1)
n2 = 0
p0
output(n2)
proc p0 {  input(n3)
           n4 = add(n3,n0)
           n2 = mul(n1,n4)  }
```

Your TASK:

IMPLEMENT and TEST the
SPL **Value Flow Checker**

And **now**...

HAPPY PAIR-**CODING**!



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