Wake-up Exercise

What does the following C++ code print?

```cpp
#include <iostream>
using namespace std;
int N = 5;

int main()
{
    static int x = 1;
    if (cout << x << " " && x++ < N && main())
    { }
    return 0;
}
```

https://ilias3.uni-stuttgart.de/vote/0ZT9
Wake-up Exercise

What does the following C++ code print?

```cpp
#include <iostream>
using namespace std;
int N = 5;

int main()
{
    static int x = 1;
    if (cout << x << " " && x++ < N && main())
    {
    }
    return 0;
}
```

Result: 1, 2, 3, 4, 5

https://ilias3.uni-stuttgart.de/vote/0ZT9
Wake-up Exercise

What does the following C++ code print?

```cpp
#include <iostream>
using namespace std;
int N = 5;

int main()
{
    static int x = 1;
    if (cout << x << " " && x++ < N && main())
    {
    }
    return 0;
}
```

Result: 1, 2, 3, 4, 5

https://ilias3.uni-stuttgart.de/vote/0ZT9

Recursive call of `main`

2 - 3
Wake-up Exercise

What does the following C++ code print?

```cpp
#include <iostream>
using namespace std;
int N = 5;

int main()
{
    static int x = 1;
    if (cout << x << " " && x++ < N && main())
    {
    }
    return 0;
}
```

Variable keeps its value across invocations of the function

Result: 1, 2, 3, 4, 5

https://ilias3.uni-stuttgart.de/vote/0ZT9
Wake-up Exercise

What does the following C++ code print?

```cpp
#include <iostream>
using namespace std;
int N = 5;

int main()
{
    static int x = 1;
    if (cout << x << " " && x++ < N && main())
    {
    }
    return 0;
}
```

Result: 1, 2, 3, 4, 5
Wake-up Exercise

What does the following C++ code print?

```cpp
#include <iostream>
using namespace std;
int N = 5;

int main()
{
    static int x = 1;
    if (cout << x << " " && x++ < N && main())
    {
    }
    return 0;
}
```

Result: 1, 2, 3, 4, 5

https://ilias3.uni-stuttgart.de/vote/0ZT9
Overview

- Introduction to Types and Type Systems
- Type Checking
  - Type Equivalence
  - Type Compatibility
  - Formal Definition of Type Systems
  - Type Inference
- Equality Testing and Assignments
Type Compatibility

- Check whether **combining two values** is **valid according to their types**
- “Combining” may mean
  - **Assignment**: Are left-hand side and right-hand side compatible?
  - **Operators**: Are operands compatible with the operator and with each other?
  - **Function calls**: Are actual arguments and formal parameters compatible?
Compatible ≠ Equal

Most PLs: Types may be **compatible**
even when **not the same**

Example (C):

```c
double d = 2.3;
float f = d * 2;
int i = f;
printf("%d\n", i);
```
Compatible ≠ Equal (2)

- Rules of PL define which types are compatible
- Examples of rules
  - Can assign subtype to supertype:
    \[ \text{lhs} = \text{rhs} ; \]
  - Different number types are compatible with each other
  - Collections of same type are compatible, even if length differs
Type Conversions

When types aren’t equal, they must be converted

- Option 1: **Cast** = explicit type conversion
  - Programmer changes value’s type from T1 to T2

- Option 2: **Coercion** = implicit type conversion
  - PL allows values of type T1 in situation where type T2 expected

- Both options: Actual conversion happens at runtime
Runtime Behavior of Conversions

What happens during conversion?

Three cases:

- Types are structurally equivalent: Conversion is only conceptual, no instructions executed.
- Types have different sets of values, but are represented in the same way in memory: May need check that value is in target type.
- Different low-level representations: Need special instructions for conversion.
Examples (Ada)

\[ n : \text{integer}; \]
\[ r : \text{long\_float}; \]
\[ t : \text{integer range} \ 0..100; \quad \text{-- has alias: test\_score} \]
\[ c : \text{celsius\_temp}; \quad \text{-- type alias for integer} \]

\[ t := \text{test\_score (n)}; \quad \text{-- runtime semantic check} \]
\[ n := \text{integer (t)}; \quad \text{-- no check needed} \]
\[ r := \text{long\_float (n)}; \quad \text{-- runtime conversion} \]
\[ n := \text{integer (r)}; \quad \text{-- runtime conversion and check} \]
\[ n := \text{integer (c)}; \quad \text{-- purely conceptual} \]
\[ c := \text{celsius\_temp (n)}; \quad \text{-- purely conceptual} \]
Coercions in C

- Most **primitive types are coerced** whenever needed
- Some coercions **may loose information**
  - float to int: Loose fraction
  - int to char: Causes char to overflow (and will give unexpected result)
- Enable compiler warnings to avoid surprises
Coercions in C

- Most primitive types are coerced whenever needed.
- Some coercions may lose information:
  - float to int: Loose fraction
  - int to char: Causes char to overflow (and will give unexpected result)
- Enable compiler warnings to avoid surprises.

Source: geeksforgeeks.org
Coercions in C: Demo

Demo: coercions.c
compile with gcc -Wconversion
Coercions in JavaScript

- **Almost all types are coerced when needed**
  - Rationale: Websites shouldn’t crash

- **Some coercions make sense:**
  - "number:" + 3 yields "number:3"

- **Many others are far from intuitive:**
  - [1, 2] << "2" yields 0

More details and examples:

*The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript.* Pradel and Sen. ECOOP 2015
Overview

- Introduction to Types and Type Systems
- Type Checking
  - Type Equivalence
  - Type Compatibility
  - Formal Definition of Type Systems
  - Type Inference
- Equality Testing and Assignments
Formally Defined Type Systems

- **Type systems are**
  - implemented in a compiler
  - formally described
  - and sometimes both

- **Active research area with dozens of papers each year**
  - Focus: New languages and strong type guarantees

- **Example here:** Typed expressions
Typed Expression: Syntax

\[ t ::= \begin{array} {l}
\text{true} \\ \text{false} \\ \text{if} \ t \ \text{then} \ t \ \text{else} \ t \\ 0 \\ \text{succ} \ t \\ \text{pred} \ t \\ \text{iszero} \ t
\end{array} \]

(semantic not formally defined)

Examples

\[ \begin{array} {l}
\text{succ} \ 0 \quad (= 1) \\
\text{if} \ (\text{iszero} \ (\text{pred} \ (\text{succ} \ 0))) \\
\text{then} \ 0 \\
\text{else} \ (\text{succ} \ 0) \quad (= 0)
\end{array} \]
Not All Expressions Make Sense

- Only **some expressions** can be evaluated
  - Other don’t make sense
  - Implementation of the language would **get stuck** or throw a **runtime error**
Types to the Rescue

- Use **types to check** whether an expression is meaningful
  - If term \( t \) has a type \( T \), then its evaluation won’t get stuck
  - Written as \( t : T \)

- **Two types**
  - \( Nat \) .. natural numbers
  - \( Bool \) .. Boolean values
Examples

\[
\text{if} \ (\text{iszero} \ 0) \ \text{then} \ \text{true} \ \text{else} \ 0 \\
\text{succ} \ (\text{if} \ 0 \ \text{then} \ \text{true} \ \text{else} \ (\text{pred} \ \text{false})) \ \\
\text{if} \ \text{true} \ \text{then} \ \text{false} \ \text{else} \ \text{true} : \ \text{Bool} \\
\text{pred} \ (\text{succ} \ (\text{succ} \ 0)) : \ \text{Nat}
\]
Type Rules

Background: \[ \frac{\text{A}}{\text{B}} \]  
\[ \text{rule} \]

if \( \text{A} \) is true then \( \text{B} \) is true

\[ \text{Bool: } \frac{\text{true: Bool}}{} \]  
\[ \text{false: Bool} \]  

\[ (\text{T-True}) \]

\[ (\text{T-False}) \]

\[ \frac{\text{t}_1: \text{Bool}}{\text{t}_2: \text{T}} \frac{\text{t}_3: \text{T}}{} \text{ (T-1f)} \]

\[ \text{if } \text{t}_1 \text{ then } \text{t}_2 \text{ else } \text{t}_3: \text{T} \]

\[ \text{Nat: } \frac{\text{0: Nat}}{} \]  
\[ \text{succ } \text{t}_0: \text{Nat} \]  

\[ (\text{T-Succ}) \]

\[ \frac{\text{t}_1: \text{Nat}}{\text{pred } \text{t}_0: \text{Nat}} \text{ (T-Pred)} \]

\[ \frac{\text{t}_0: \text{Nat}}{\text{iszero } \text{t}_0: \text{Bool}} \text{ (T-Iszero)} \]
Type Checking Expressions

- **Typing relation**: Smallest binary relation between terms and types that satisfies all instances of the rules
- Term $t$ is **typable (or well typed)** if there is some $T$ such that $t : T$
- **Type derivation**: Tree of instances of the typing rules that shows $t : T$
Type Derivations: Example

true: Bool
false: Bool
true: Bool
if true then false else true : Bool
Can't apply any axiom or rule. Expression not well typed!

\[
\begin{align*}
0 & : \text{Bool} \\
\text{true} & : \text{Nat} \\
(\text{pred false}) & : \text{Nat} \\
\text{if } 0 \text{ then true else } (\text{pred false}) & : \text{Nat} \\
\text{succ } (\text{if } 0 \text{ then true else } (\text{pred false})) & : \text{Nat}
\end{align*}
\]
Quiz: Typing Derivation

Find the typing derivation for the following expression:

\[
\text{if false then } \text{(pred(pred 0)) else (succ 0)}
\]

How many axioms and rules do you need?
Quiz:

3 axioms, 4 rules

\[
\begin{align*}
&\text{false : Bool} \\
&\text{pred } 0 : \text{Nat} \\
&\text{pred (pred } 0\text{)} : \text{Nat} \\
&\text{if false then } (\text{pred (pred } 0\text{)}) \text{ else } (\text{succ } 0) : \text{Nat}
\end{align*}
\]
Overview

- Introduction to Types and Type Systems

- Type Checking
  - Type Equivalence
  - Type Compatibility
  - Formal Definition of Type Systems
  - Type Inference

- Equality Testing and Assignments
Some PLs are **statically typed** but allow programmers to **omit some type annotations**

- Get **guarantees** of static type checking
- Without paying the cost of full type annotations
- Different from gradual typing, where programmer decides when and where to annotate types
Example

// Scala
var businessName = "Montreux Jazz Cafe"

def squareOf(x: Int) = x * x

businessName = squareOf(23)
Inferred to be a String

// Scala
var businessName = "Montreux Jazz Cafe"

def squareOf(x: Int) = x * x

businessName = squareOf(23)
Example

```scala
// Scala
var businessName = "Montreux Jazz Cafe"

def squareOf(x: Int) = x * x

businessName = squareOf(23)
```

Inferred to be a String

```
Inferred to return an Int
```

Compile-time type error:
Can’t assign Int to String variable
Unifying Partial Type Information

■ **Type inference algorithm:**
  Unifies partial type information for two values whenever type rules expect them to be the same

■ **Type checking:**
  Find unique type for each value with no contradictions and no ambiguous occurrences of overloaded names
Overview

■ Introduction to Types and Type Systems

■ Type Checking
  □ Type Equivalence
  □ Type Compatibility
  □ Formal Definition of Type Systems
  □ Type Inference

■ Equality Testing and Assignments
Equality Testing

- **Equality operator:** Many possible meanings
  - E.g., for two strings
    - Are the strings aliases?
    - Are they bitwise identical?
    - Do they have the same sequence of chars?
    - Do they look the same if printed?
- **Meaning depends on PL and types of values**
Assignments

- **Assignment operator**: Multiple possible meanings
- **Deep vs. shallow**
  - (Deep) copy of right-hand side to left-hand side’s location
  - Mostly used for primitive types
  - Rare for objects, but useful, e.g., for remote procedure calls
- Copy reference to right-hand side to left-hand side
- Mostly used for objects
Quiz: Types

Which of the following statements is true?

- Types are compatible if and only if they are equal
- Coercions mean that a programmer casts a value from one type to another type
- Type conversions are guaranteed to preserve the meaning of a value
- PLs with type inference may provide static type guarantees

https://ilias3.uni-stuttgart.de/vote/0ZT9
Quiz: Types

Which of the following statements is true?

- Types are compatible if and only if they are equal
- Coercions mean that a programmer casts a value from one type to another type
- Type conversions are guaranteed to preserve the meaning of a value
- PLs with type inference may provide static type guarantees

https://ilias3.uni-stuttgart.de/vote/0ZT9
Overview

■ Introduction to Types and Type Systems

■ Type Checking
  □ Type Equivalence
  □ Type Compatibility
  □ Formal Definition of Type Systems
  □ Type Inference

■ Equality Testing and Assignments
Overview

- Introduction to Types and Type Systems
- Type Checking
  - Type Equivalence
  - Type Compatibility
  - Formal Definition of Type Systems
  - Type Inference
- Equality Testing and Assignments