Eager Parameter Evaluation

Sometimes we want parameters evaluated eagerly—as soon as they are known.

Consider a sorting routine that breaks an array in half, sorts each half, and then merges together the two sorted halves (this is a merge sort).

In outline form it is:

```c
sort(inputArray) {
  ...
  merge(sort(leftHalf(inputArray)),
  sort(rightHalf(inputArray)));}
```

This definition lends itself nicely to parallel evaluation: The two halves of an input array can be sorted in parallel. Each of these two halves can again be split in two, allowing parallel sorting of four quarter-sized arrays,
then leading to 8 sorts of 1/8 sized arrays, etc.

But, to make this all work, the two parameters to merge must be evaluated eagerly, rather than in sequence.
Type Equivalence

Programming languages use types to describe the values a data object may hold and the operations that may be performed.

By checking the types of values, potential errors in expressions, assignments and calls may be automatically detected. For example, type checking tells us that

\[ 123 + "123" \]

is illegal because addition is not defined for an integer, string combination.

Type checking is usually done at compile-time; this is static typing.

Type-checking may also be done at run-time; this is dynamic typing.
A program is **type-safe if it is impossible** to apply an operation to a value of the wrong type. In a type-safe language, `plus` is never told to add an integer to a string, because its definition does not allow that combination of operands. In type-safe programs an operator can still see an illegal value (e.g., a division by zero), but it can’t see operands of the wrong type.

A **strongly-typed programming language** forbids the execution of type-unsafe programs.

Weakly-typed **programming languages** allow the execution of potentially type-unsafe programs.
The question reduces to whether the programming language allows programmers to “break” the type rules, either knowingly or unknowingly.

Java is strongly typed; type errors preclude execution. C and C++ are weakly typed; you can break the rules if you wish. For example:

```c
int i; int* p;
p = (int *) i * i;
```

Now \( p \) may be used as an integer pointer though multiplication need not produce valid integer pointers.
If we are going to do type checking in a program, we must decide whether two types, T1 and T2 are equivalent; that is, whether they be used interchangeably.

There are two major approaches to type equivalence:

Name Equivalence:

Two types are equivalent if and only if they refer to exactly the same type declaration.

For example,

```plaintext
type PackerSalaries = int[100];
type AssemblySizes = int[100];
PackerSalaries salary;
AssemblySizes size;
```
Is 

\[
sal = \text{size};
\]

allowed?

Using name equivalence, no. That is, salary \(\not\equiv_N\) size since these two variables have different type declarations (that happen to be identical in structure).

Formally, we define \(\equiv_N\) (name type equivalence) as:

(a) \(T \equiv_N T\)

(b) Given the declaration

\[
\text{Type } T_1 = T_2;
\]

\(T_1 \equiv_N T_2\)
We treat anonymous types (types not given a name) as an abbreviation for an implicit declaration of a new and unique type name.

Thus

```java
int A[10];
```

is an abbreviation for

```java
Type T_new = int[10];
T_new A;
```