Interpretation and Compilation of Languages

Master Programme in Computer Science

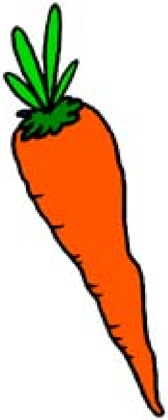
Mário Pereira   mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

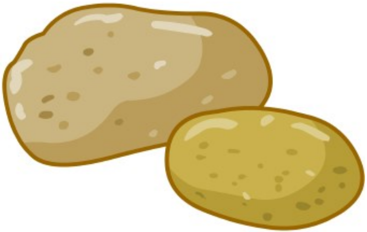
April 14, 2025

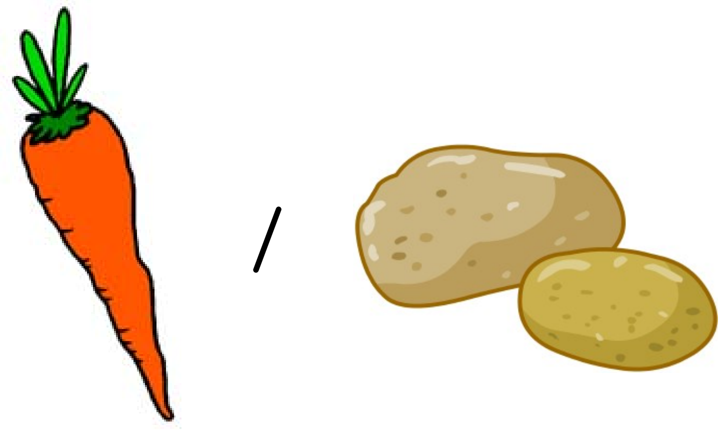
Lecture 6

based on lectures by Jean-Christophe Filliâtre and Léon Gondelman
previous editions by João Costa Seco, Luís Caires, and Bernardo Toninho



+





If we write

```
    "5" + 37
```

do we get

- a compile-time error? (OCaml, Rust, Go)
- a runtime error? (Python, Julia)
- the integer 42? (Visual Basic, PHP)
- the string "537"? (Java, Scala, Kotlin)
- a pointer? (C, C++)
- something else?

And what about

```
  37 / "5"
```

?

If we now add two arbitrary expressions

```
    e1 + e2
```

How can we decide whether this is legal and which operation to perform?

The answer is typing, a program analysis that binds types to each sub-expression, to rule out inconsistent programs.

But what does *inconsistent* really means? Simply, a program for which there is no semantics.

(in close connection with Lecture 2)

When?

Some languages are dynamically typed: types are bound to values and are used at runtime

examples: Lisp, PHP, Python, Julia

Other languages are statically typed: types are bound to expressions and are used at compile time

examples: C, C++, Java, OCaml, Rust, Go

Consider the following C and Python code snippets:

```c
int id(int num) {
  return num; }

void main(){
  printf("%d", id(42,42));}
```

```python
def id(num):
  return num

print(id(42,42))
```

The C code fails at the compile-time (compilation error)
error: too many arguments to function 'id'

The Python code compiles to the VM and fails at runtime (runtime error)
TypeError: id() takes 1 positional argument but 2 were given

Remark

A language may use both static and dynamic typing.

We will illustrate it with JAVA at the end of this lecture.

Today: Static Typing

1. type checking
   - examples
   - type checking WHILE, formally
   - type safety lemma
2. implementing a type checker
3. subtyping
4. overloading

Static Typing

- Type checking must be decidable.
- Type checking must reject programs whose evaluation would fail; this is type safety.
- Type checking must not reject too many non-absurd programs; the type system must be expressive.

Example

```c
int f(long x) { return *x; }
```

```
file.c:1:23: error: invalid type argument of unary ‘*’
                (have ‘long int’)
    2 | int f(long x) { return *x; }
      |                        ^~
```

But...

On the other hand, the compiler accepts an integer where a pointer is expected

```c
int *f(long x) { return x; }
```

Yet it emits a warning

```
file.c:1:24: warning: returning ‘long int’ from a function
             with return type ‘int *’ makes pointer from
             integer without a cast [-Wint-conversion]
    1 | int *f(long x) { return x; }
      |                         ^
```

One can “bypass” type checking with a cast.

```
int f(long x) { return *((int*)x); }
```

and Many Other Things

Type checking also means checking

- the existence of types, of structure fields, etc.
- existence and uniqueness of functions, their arity
- existence of variables
- that a `break` is within a loop
- that `&e` mentions a left value
- etc.

1. Any sub-expression is annotated with a type

```
int f(int x) { int y = ((x:int)+(1:int):int); ... }
```

   type checking is easy but this is unmanageable for the programmer

2. Only annotate variable declarations (C, C++, Java, etc.)

```
int f(int x) { int y = x+1; return y; }
```

3. only annotate function parameters (C++ 11, Java 10)

```
int f(int x) { var y = x+1; return y; }
```

4. No annotation at all ⇒ type inference (OCaml, Haskell, etc.)

```
fun x -> x+1
```

Limits of Static Typing

By nature, verification performed by static typing are limited to decidable properties.

(reminder: *decidable* means that we can write a program that, for any input, terminates and outputs *yes* or *no*).

Almost all “interesting” properties over source code are not decidable (Rice theorem).

The compiler cannot check

- the absence of division by zero
- that a computation terminates
- the absence of arithmetic overflow
- etc.

(and beyond that a program is doing what it is supposed to do!)

Yet, it is possible

- to detect errors with certainty in some cases
- to emit warnings, at the risk of false positives

Let us consider the language WHILE from lecture 2.

We are going to make it (even) more similar to C

- memory allocation
- read/write from/to memory

We are going to formally

1. give type checking rules for this language
2. show a type safety property

e	$::=$		**expression**
	$\mid$	v	integer or Boolean constant
	$\mid$	x	variable
	$\mid$	$e \oplus e$	binary operator $(+, <, \ldots)$
	$\mid$	$*e$	read from memory
s	$::=$		**statement**
	$\mid$	x `:=` e	assignment
	$\mid$	x `:=malloc(4)`	allocate memory
	$\mid$	`*`e`:=`e	write to memory
	$\mid$	`if` `(`e`)` s `else` s	conditional
	$\mid$	`while` `(`e`)` s	loop
	$\mid$	s `;` s	block

(values are on the next slide)

The notion of value from lecture 2 is updated

$$
\begin{array}{rcll}
v & ::= & & \textbf{value} \\
  & | & n & \text{integer value} \\
  & | & b & \text{Boolean value} \\
  & | & \ell & \text{memory address}
\end{array}
$$

We define a big-step operational semantics for expressions

$$\mu, \sigma, e \Downarrow v$$

and a small-step operational semantics for statements

$$\mu, \sigma, s \to \mu', \sigma', s'$$

where σ is a function from variables to values
and μ is a function from addresses (ℓ) to integers (n) or Booleans (b).

Big-step operational semantics is suitable to define interpreters.

It is very easy to implement a recursive function that evaluates sub-expressions and sub-commands under the right environment. Example:

$$\frac{\sigma, s_1 \Downarrow \sigma_1 \quad \sigma_1, s_2 \Downarrow \sigma_2}{\sigma, s_1 \,;\, s_2 \Downarrow \sigma_2}$$

For the sequence case, the interpreter

1. recursively evaluates s_1 under σ, getting σ_1
2. recursively evaluates s_2 under σ_1, getting σ_2

This is exactly what we did in Lecture 2.

Small-step operational semantics is suitable to analyze individual steps of computation.

This is crucial when doing typing analysis. We want to make sure every step of computation preserves the same type. Example:

$$\frac{}{\sigma, \texttt{skip}\,;\, s \rightarrow \sigma, s} \qquad \frac{\sigma, s_1 \rightarrow \sigma_1, s_1'}{\sigma, s_1\,;\, s_2 \rightarrow \sigma_1, s_1'\,;\, s_2}$$

In small-step semantics, when the command is a sequence $s_1\,;\, s_2$:

- if $s_1 = \texttt{skip}$, then reduce $s_1\,;\, s_2$ to s_1 and do not change the environment
- if s_1 is not $\texttt{skip}$, then
  1. take one reduction step from s_1 to s_1'
  2. if needed, update σ into σ_1
  3. continue evaluation of $s_1'\,;\, s_2$ under σ_1

$$\frac{}{\mu, \sigma, n \Downarrow n} \qquad \frac{x \in \mathtt{dom}(\sigma)}{\mu, \sigma, x \Downarrow \sigma(x)}$$

$$\frac{\mu, \sigma, e_1 \Downarrow n_1 \quad \mu, \sigma, e_2 \Downarrow n_2 \quad n \stackrel{\text{def}}{=} n_1 + n_2}{\mu, \sigma, e_1 + e_2 \Downarrow n} \qquad \text{etc.}$$

$$\frac{\mu, \sigma, e \Downarrow \ell \quad \ell \in \mathtt{dom}(\mu)}{\mu, \sigma, *e \Downarrow \mu(\ell)}$$

Note: expressions do not have side-effects and always terminate.
That is why use big-step semantics.

$$\frac{\mu, \sigma, e \Downarrow v}{\mu, \sigma, x\texttt{:=}e \rightarrow \mu, \sigma\{x \mapsto v\}, \texttt{skip}} \qquad \frac{\mu, \sigma, e_1 \Downarrow \ell \quad \ell \in \texttt{dom}(\mu) \quad \mu, \sigma, e_2 \Downarrow n}{\mu, \sigma, \texttt{*}e_1\texttt{:=}e_2 \rightarrow \mu\{\ell \mapsto n\}, \sigma, \texttt{skip}}$$

$$\frac{\ell \notin \texttt{dom}(\mu)}{\mu, \sigma, x\texttt{:=malloc(4)} \Downarrow \mu\{\ell \mapsto n\}, \sigma, \texttt{skip}}$$

$$\frac{}{\mu, \sigma, \texttt{skip;}\, s \rightarrow \mu, \sigma, s}$$

$$\frac{\mu, \sigma, s_1 \rightarrow \mu_1, \sigma_1, s_1'}{\mu, \sigma, s_1\,\texttt{;}\, s_2 \rightarrow \mu_1, \sigma_1, s_1'\,\texttt{;}\, s_2}$$

$$\frac{\mu, \sigma, e \Downarrow \texttt{true}}{\mu, \sigma, \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2 \rightarrow \mu, \sigma, s_1} \qquad \frac{\mu, \sigma, e \Downarrow \texttt{false}}{\mu, \sigma, \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2 \rightarrow \mu, \sigma, s_2}$$

$$\frac{\mu, \sigma, e \Downarrow \texttt{true}}{\mu, \sigma, \texttt{while (}e\texttt{)}\ s \rightarrow \mu, \sigma, s\texttt{;}\,\texttt{while (}e\texttt{)}\ s}$$

$$\frac{\mu, \sigma, e \Downarrow \texttt{false}}{\mu, \sigma, \texttt{while (}e\texttt{)}\ s \rightarrow \mu, \sigma, \texttt{skip}}$$

We introduce types, with the following abstract syntax

$$
\begin{array}{rcll}
\tau & ::= & & \textbf{type} \\
 & | & \texttt{int} & \text{type of integers} \\
 & | & \texttt{int}* & \text{type of pointers}
\end{array}
$$

The type of a variable is given by a typing environment Γ
(a function from variables to types)

The typing judgment/relation is written

$$\Gamma \vdash e : \tau$$

and reads "in typing environment Γ, expression e has type τ"

We use inference rules to define $\Gamma \vdash e : \tau$

$$\frac{}{\Gamma \vdash n : \mathtt{int}}$$

$$\frac{x \in \mathtt{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{int} \quad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash e_1 + e_2 : \mathtt{int}}$$

etc.

$$\frac{\Gamma \vdash e : \mathtt{int*}}{\Gamma \vdash \mathtt{*}e : \mathtt{int}}$$

With $\Gamma = \{p \mapsto \texttt{int*};\ b \mapsto \texttt{int}\}$, we have

$$\dfrac{\dfrac{\dfrac{p \in \texttt{dom}(\Gamma)}{\Gamma \vdash p : \texttt{int*}}}{\Gamma \vdash *p : \texttt{int}} \qquad \dfrac{b \in \texttt{dom}(\Gamma)}{\Gamma \vdash b : \texttt{int}}}{\Gamma \vdash *p + b : \texttt{int}}$$

This derivation is a proof that `*p+b` is `well-typed`.

In the same environment, we cannot type expressions such as

$$*p + c$$

or

$$*42$$

or

$$1 + p$$

This is precisely what we want, for these expressions have no semantics.

To type statements, we introduce a new judgment

$$\Gamma \vdash s$$

that reads "in environment Γ, statement s is well-typed".

Type Checking Statements

$$\frac{x \in \mathrm{dom}(\Gamma) \quad \Gamma(x) = \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash x \texttt{:=} e}$$

$$\frac{x \in \mathrm{dom}(\Gamma) \quad \Gamma(x) = \texttt{int*}}{\Gamma \vdash x \texttt{:=malloc(4)}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int*} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{*}e_1 \texttt{:=} e_2}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \texttt{if}\ \texttt{(}e\texttt{)}\ s_1\ \texttt{else}\ s_2}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash s}{\Gamma \vdash \texttt{while}\ \texttt{(}e\texttt{)}\ s}$$

$$\frac{}{\Gamma \vdash \texttt{skip}} \qquad \frac{\Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash s_1\,;\,s_2}$$

well-typed programs do not go wrong

(Milner, 1978)

Type safety means evaluation won't be stuck on any expression such as

$$*42$$

or on a statement, such as

$$\texttt{if}\,(e)\ s_1\ \texttt{else}\ s_2$$

where e does not evaluate to a value.

What does stuck mean?
That we are not able to derive the semantics of such program!

Static typing is strong:

we want to know if a program has semantics,

without actually running it.

Hence, type checking process is a compromise between

- expressiveness
- decidability

This is a big take-home message.

The example of `if..else` statements is paradigmatic.

Operational semantics:

$$\frac{\mu, \sigma, e \Downarrow \texttt{true}}{\mu, \sigma, \texttt{if}\,(e)\ s_1\ \texttt{else}\ s_2 \rightarrow \mu, \sigma, s_1} \qquad \frac{\mu, \sigma, e \Downarrow \texttt{false}}{\mu, \sigma, \texttt{if}\,(e)\ s_1\ \texttt{else}\ s_2 \rightarrow \mu, \sigma, s_2}$$

How many rules do you see?

Static typing:

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \texttt{if}\,(e)\ s_1\ \texttt{else}\ s_2}$$

How many rules do you see?

This means that the following program

```
if (false) *42 else 73
```

has semantics: it always returns 73, under any environment. But ...

... it is not well-typed. The first branch does not have a type.

If we want to be correct, complete, and still decidable,
we must make some expressiveness compromises.

Type Safety – Formally

Let us show that our type system is safe wrt our operational semantics.

Theorem (type safety)

If $\Gamma \vdash s$, then the reduction of s is either infinite or reaches `skip`.

Or, equivalently,

Theorem

If $\Gamma \vdash s$ and $\mu, \sigma, s \xrightarrow{\star} \mu', \sigma', s'$ and s' is irreducible, then s' is `skip`.

Here, $\xrightarrow{\star}$ means "many reduction steps".

We will break this big Theorem into smaller auxiliary Lemmas.

Definition (well-typed environment)

An execution environment μ, σ is well-typed in a typing environment Γ, written $\Gamma \vdash \mu, \sigma$, if

$$\forall x,\ \textit{if}\ \Gamma(x) = \tau\ \textit{then}\ \begin{cases} x \in \mathtt{dom}(\sigma)\ \textit{and}\ \Gamma \vdash \sigma(x) : \tau, \\ \textit{if}\ \tau = \mathtt{int*}\ \textit{then}\ \sigma(x) \in \mu. \end{cases}$$

Lemma (evaluation of a well-typed expression)

If $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash \mu, \sigma$*, then* $\mu, \sigma, e \Downarrow v$ *and* $\Gamma \vdash v : \tau$*.*
Beside, if $v = \ell$ *then* $\ell \in \mu$*.*

Proof: by induction on the derivation $\Gamma \vdash e : \tau$.

The type safety proof is based on two lemmas

Lemma (progress)

If $\Gamma \vdash s$ and $\Gamma \vdash \mu, \sigma$, then either s is `skip`*, or $\mu, \sigma, s \rightarrow \mu', \sigma', s'$.*

Proof: by induction on the derivation $\Gamma \vdash s$.

Lemma (preservation)

If $\Gamma \vdash s$, if $\Gamma \vdash \mu, \sigma$ and if $\mu, \sigma, s \rightarrow \mu', \sigma', s'$ then $\Gamma \vdash s'$ and $\Gamma \vdash \mu', \sigma'$.

Proof: by induction on the derivation $\Gamma \vdash s$.

Now we can deduce type safety easily

Theorem (type safety)

If $\Gamma \vdash s$ and $\Gamma \vdash \mu, \sigma$ and $\mu, \sigma, s \xrightarrow{\star} \mu', \sigma', s'$ and s' is irreducible, then s' is `skip`.

proof: we have $\mu, \sigma, s \to \mu_1, \sigma_1, s_1 \to \cdots \to \mu', \sigma', s'$ and by repeated applications of the preservation lemma, we have $\Gamma \vdash s'$
by the progress lemma, s' is reducible or is `skip`
so this is `skip`.

Languages such as JAVA or OCAML enjoy such a type safety property.

Which means that the evaluation of an expression of type τ

- either does not terminate
- or raises an exception
- or terminates on a value with type τ

In OCaml, the absence of `null` makes it a rather strong property.

Implementing a Type Checker

There is a difference between the typing rules, which define the relation

$$\Gamma \vdash e : \tau$$

and the type checking algorithm, which checks that a given expression e is well-typed in some environment Γ.

For instance

- the type τ is not necessarily given (type inference)
- several rules may apply for a single construct
- an expression may have several types

The case of WHILE is simple, as a single rule applies for each expression.

We say that typing is syntax-directed.

Type checking is then implemented with a linear time traversal of the program abstract syntax tree.

- We do not simply say

```
type error
```

  but we explain the type error precisely.

- We keep types for the further phases of the compiler.

To do this, we decorate abstract syntax trees

- input of type checking contains positions in source code
- output of type checking contains types

source file	$\xrightarrow{\text{parser}}$	syntax trees with locations	$\xrightarrow{\text{type checker}}$	syntax trees with types
`t[i]`		`*(t+i)`		`*(t+4i)`

In OCaml

```
type loc = ...
```

```
type expr =




| Evar   of string
| Econst of int
| Efield of expr * string
...
```

In Java

```
class Loc { ... }
```

```
abstract class Expr {



}
class Evar   extends Expr {...}
class Econst extends Expr {...}
class Efield extends Expr {...}
...
```

In OCaml

```ocaml
type loc = ...
```

```ocaml
type expr = {
  desc: desc;
  loc : loc;
}
and desc =
| Evar   of string
| Econst of int
| Efield of expr * string
...
```

In Java

```java
class Loc { ... }
```

```java
abstract class Expr {
  Loc loc;


}
class Evar   extends Expr {...}
class Econst extends Expr {...}
class Efield extends Expr {...}
...
```

We signal a type error with an exception.

The exception contains

- a message explaining the error
- a position in the source code

We catch this exception in the main file of the compiler.

We display the position and the message

```
test.c:8:14: error: too few arguments to function 'f'
```

Output

We set up an abstract syntax for types

```ocaml
type typ = ...
```

```java
class Typ { ... }
```

and a new abstract syntax for programs

```ocaml
type texpr = {
  tdesc: tdesc;
  typ  : typ
}
and tdesc =
| Tvar   of string
| Tconst of int
| Tfield of texpr * string
...
```

```java
abstract class Texpr {
  Typ typ;


}
class Tvar   extends Texpr {...}
class Tconst extends Texpr {...}
class Tfield extends Texpr {...}
...
```

The type checker turns a parsed syntax tree into another, typed syntax tree

$$\boxed{\text{parsed trees}} \xrightarrow{\text{type checker}} \boxed{\text{typed trees}}$$

We build new trees. Yet this is efficient, since

- it is typically a linear traversal
- former AST are collected by the Garbage Collector

At any moment, we know the variables that are in scope

For each one, we know

- the location of its declaration
- its type
- possibly other things (size in memory, etc.)

Terminology: symbol tables refer to such data associated to symbols inside compilers.

Subtyping

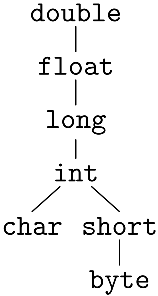
We say that a type τ_1 is a subtype of a type τ_2, which we write

$$\tau_1 \leq \tau_2$$

if any value with type τ_1 can be considered as a value with type τ_2.

In many languages, there is subtyping between numerical types.

In JAVA, it is as shown on the right



Thus we can write

```java
int n = 'a';
```

but not

```java
byte b = 144;
```

In an object-oriented language, inheritance induces subtyping:
if a class `B` inherits from a class `A`, we have

$$\mathtt{B} \leq \mathtt{A}$$

i.e., any value of type `B` can be seen as a value of type `A`.

The two classes

```
class Vehicle               { ... void move() { ... } ... }
class Car extends Vehicle { ... void move() { ... } ... }
```

induce the subtyping relation

$$\mathtt{Car} \leq \mathtt{Vehicle}$$

and thus we can write

```
Vehicle v = new Car();
v.move();
```

The statement `new C(...)` builds an object of class `C`, and the class of this object cannot be changed in the future; this is the dynamic type of the object.

However, the static type of an expression, as computed by the compiler, may differ from the dynamic type, because of subtyping.

When we write

```
Vehicle v = new Car();
v.move();
```

variable `v` has type `Vehicle`, but the method `move` that is called is that of class `Car`.

In many cases, the compiler cannot determine the dynamic type.

Example:

```java
void moveAll(LinkedList<Vehicule> l) {
  for (Vehicule v: l)
    v.move();
}
```

Sometimes we need to force the compiler's hand, which means we claim that a value has some type.

We call this type casting (or simply `cast`).

JAVA's notation, inherited from C, is

$$(\tau)e$$

It means: "the static type of this expression is τ".

Using a cast, we can write

```
int  n = ...;
byte b = (byte)n;
```

In this case, there is no dynamic verification
(if the integer is too large, it is truncated).

Let us consider

$$(C)e$$

where

- D is the dynamic type of (the object designated by) e
- E is the static type of expression e

There are three cases

- C is a super class of E: this is an upcast and the code for $(C)e$ is that of e (but the cast has some influence anyway, since $(C)e$ has type C)
- C is a subclass of E: this is a downcast and the code contains dynamic test to check that D is indeed a subclass of C
- C is neither a subclass nor a super of E: the compiler rejects the program with a type error

```
class A {
  int x = 1;
}

class B extends A {
  int x = 2;
}
```

```
  B b = new B();
  System.out.println(b.x);      // 2
  System.out.println(((A)b).x); // 1
```

```java
void m(Vehicle v, Vehicle w) {
  ((Car)v).await(w);
}
```

Nothing guarantees that the object passed to `m` will be a car; in particular, it could have no method `await`!

A dynamic test is required.

JAVA raises `ClassCastException` is the test fails.

To allow defensive programming, there exists a Boolean construct

$$e \ \texttt{instanceof} \ C$$

that checks whether the class of e is indeed a subclass of C.

It is idiomatic to do

```
if (e instanceof C) {
  C c = (C)e;
  ...
}
```

In this case, the compiler makes an optimization to perform a single test.

Overloading

Overloading is the ability to reuse the same name of several operations.

Overloading is handled at compile time, using the number and the (static) types of arguments.

Caveat: not to be confused with *overriding* (inheritance).

Example

In JAVA, operation + is overloaded

```
int    n = 40 + 2;
String s = "foo" + "bar";
String t = "foo" + 42;
```

These are three distinct operations

```
int    +(int   , int   )
String +(String, String)
String +(String, int   )
```

Be Careful!

When we write

```
int n = 'a' + 42;
```

this is subtyping that allows us to consider `'a'` with type `char` as a value of type `int`, and thus the operation is `+(int, int)`. It is overriding.

But when we write

```
String t = "foo" + 42;
```

this is not subtyping ($\texttt{int} \not\leq \texttt{String}$). It is overloading :)

In particular, we cannot write

```
String t = 42;
```

Another Example

In JAVA, one cannot overload operators such as +
but one can overload methods/constructors

```
int f(int n, int m) { ... }
int f(int n)        { ... }
int f(String s)     { ... }
```

This is exactly as if we had written

```
int f_int_int(int n, int m) { ... }
int f_int    (int n)        { ... }
int f_String (String s)     { ... }
```

The compiler uses the static types of `f`'s arguments to determine which method to call.

Yet, overloading resolution can be tricky

```
class A {...}
class B extends A {
  void m(A a) {...}
  void m(B b) {...}
}
```

with

```
{ ... B b = new B(); b.m(b); ... }
```

both methods apply.

Here, method `m(B b)` that is called, because it is considered more precise

Ambiguity

Some cases are ambiguous

```
class A {...}
class B extends A {
  void m(A a, B b) {...}
  void m(B b, A a) {...}
}
{ ... B b = new B(); b.m(b, b); ... }
```

and reported as such

```
test.java:13: reference to m is ambiguous,
  both method m(A,B) in B and method m(B,A) in B match
```

- Type checking a fragment of C.
- Covers type-checking structure pointers and function declarations.
- Lexer and parser are given.

Bibliography

- Compilers, Principles, Techniques, and Tools, Alfred Aho and Monica S. Lam and Ravi Sethi and Jeffrey D. Ullman, Second Edition[Chapter 4.7], Addison-Wesley (2006) (*“the dragon book”*).
- Types and Programming Languages, Benjamin Pierce, The MIT Press.