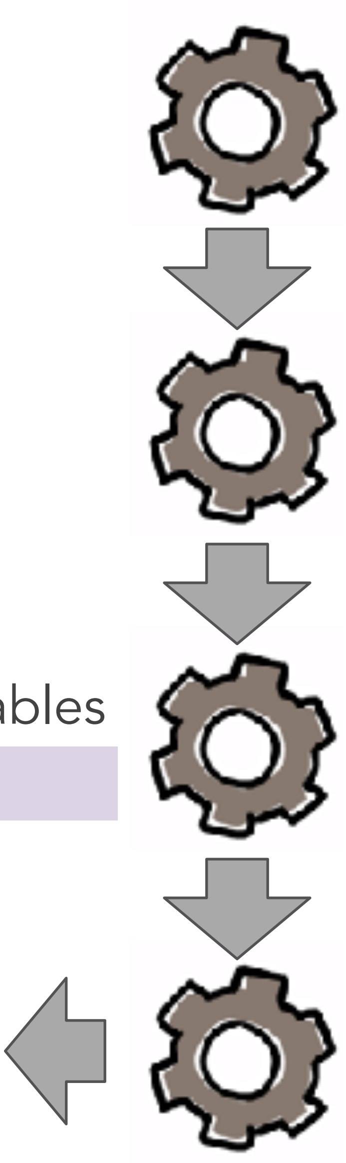
ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

SEMANTIC ANALYSIS

COSIMO LANEVE
cosimo.laneve@unibo.it

CORSO 72671
COMPLEMENTI DI LINGUAGGI DI PROGRAMMAZIONE

THIS LECTURE



lexical analysis

syntactic analysis

semantic analysis

bytecode generation

- scopes and symbol tables
- type checking

the `SimpLan` interpreter

OUTLINE

- the design of types
- environments for type checking
- type checking of expressions, statements, functions, programs
- advanced type checking
- typing effects

reference:

- Torben Morgensen: **Basics of Compiler Design**, Chapter 6

TYPES AND TYPE SYSTEMS

Definition: type

A type is

- a set of values
- a set of operations on those values

example: classes are one instance of the modern notion of type

why we use types?

- most operations are **legal** only for values of some types
- it **doesn't make sense** to add a function pointer and an integer in **C**
- it **does make sense** to add two integers
- but both have the same assembly language implementation!

example: what is the type of `addi $r1,$r2,$r3`

TYPES AND TERMINOLOGIES

type safety: a language is **type-safe** if the only operations that can be performed on data whose type is that of the operation

- type enforcement can be **static**, catching potential errors at compile time,
- or **dynamic**, associating type information with values at run-time and consulting them as needed to detect imminent errors,
- or a **combination** of both.

type system is a formal system consisting of a set of rules that assigns a property called a type to the operations of a program

- **used statically** = done at compile time
- **used dynamically** = done at runtime; it associates each runtime object with a *type tag* containing type information that can also be used to implement downcasting, reflection, etc.
- combinations… (and there are also **untyped** languages: `Python`, `JavaScript`)

type expressions: data types can be defined by programmers (structured data types)

- **type compliance & equivalence**: when two types are equal? (nominal vs structural)

TYPE CHECKING

type checking is the process of verifying that operations are used with the correct types — it may be either **static** or **dynamic**

- **type errors** arise when operations are performed on values that do not support that operation
- type checking can detect certain important kinds of errors
  - **memory errors**: reading from an invalid pointer, etc.
  - **violation of abstraction boundaries**

```
class FileSystem {
      private File open(String x){
           . . .
       }
       . . .
}
class Client {
   void f(FileSystem fs){
       File fdesc = fs.open("foo")
       . . .
   } // f cannot see inside FileSystem !
}
```

TYPES AND TERMINOLOGIES

a table for mainstream languages

TYPE SYSTEMS

Language	Type Safety	Type Expr.	Type Comp. & Equiv.	Type Checking
C	weak	explicit	nominal	checking/inference
C#	weak	implicit/explicit	nominal	checking/inference
F#	strong	implicit	nominal	inference
Go	strong	implicit/explicit	structural	inference
Haskell	strong	implicit/explicit	nominal	inference
Java	strong	explicit	nominal	checking/inference
JavaScript	weak	implicit	no	dynamic
OCaml	strong	implicit/explicit	nominal	inference
Prolog	no	no	no	dynamic
Python	strong	implicit/explicit	no	dynamic
Scala	strong	implicit	nominal/structural	checking/inference

FALSE POSITIVES AND FALSE NEGATIVES

assume there is a code for **STATIC** typechecking: `TypeCheck(P)`

- `TypeCheck(P)` takes in input a program `P`
- it returns `true` if the program is correct wrt types
- `false` otherwise

false positives: `TypeCheck(P) = true` and when you execute `P`, the execution terminates with a `program error` due to `types`

false negatives: `TypeCheck(P) = false` and when you execute `P`, the execution never shows up a `type error`

- example: `int x = 0; if (true) x = 1; else x = true;`

FALSE POSITIVES ARE PROBLEMATIC!

STATIC TYPE CHECKING: THE FORMALISM

there is no standard tool for type checkers

- they need to be written in a general-purpose programming language

there are standard notations that can be used for specifying the rules of the type checker

- they can be easily converted to code in any host language

the most common notation is **inference rules**

Definition: inference rule

An inference rule has a set of premises $J_1, \ldots, J_n$ and a conclusion J, conventionally separated by a line:

$$\frac{J_1 \quad \ldots \quad J_n}{J}$$

when the set of premises is empty, the rule is called **axiom**

TYPE CHECKING: THE FORMALISM

the inference rule

$$\frac{J_1 \quad \ldots \quad J_n}{J}$$

is read `if the Premises` $J_1, \ldots, J_n$ `are true`
`then Conclusion J is true`

- the symbols $J_1, \ \ldots, \ J_n, \ J$ are called **judgments**
- the most common judgment is $\vdash e : T$ that is read "expression e has type T"
- an example:

$$\frac{\vdash e_1 : \texttt{bool} \qquad \vdash e_2 : \texttt{bool}}{\vdash e_1 \ \&\& \ e_2 \ : \ \texttt{bool}}$$

that is read: if e_1 and e_2 have type `bool`, then $e_1 \ \&\& \ e_2$ has type `bool`

TYPE SYSTEM AND TYPE CHECKING

the set of inference rules for the types of a language is called **type system**

inference rules for language constructs can be implemented by means of **recursive functions** on the abstract syntax trees

the function `typeCheck`:

- given a term `e`, return a type `T`, such that $\vdash$ `e:T`

example:

```
Type typeCheck(Exp e) =
       switch (e){
          case a && b:
               t1 = typeCheck(a) ;
               t2 = typeCheck(b) ;
               if ((t1 == bool) && (t2 == bool))
                   return bool ;
                else break ;
       }
    }
```

the `typeCheck` method in `SimpLan` has no argument because the infos are in the fields of the object

CONTEXT AND ENVIRONMENT

how do we type-check variables?

- variables, such as `x`, can have any of the types available in a programming language
- the type it has in a particular program depends on the **context**

the context is defined by declarations that bind variables to their type

it is **a data structure** where one can **look up** a variable and **get its type**

formally, constexts are environments Γ

- see previous slides

$\Gamma \vdash e{:}\mathtt{bool}$ is read: `e` has type `bool` in the environment Γ

in the compilers, the environment is implemented by symbol tables

A TYPE SYSTEM FOR SIMPLE EXPRESSIONS

```
exp     :   NUM | ID | 'true' | 'false'
            | exp '+' exp | exp '==' exp     ;
```

the type system contains the rules (for type checking)

$$\frac{}{\Gamma \vdash \mathtt{num:\ int}}\,[\mathtt{Num}] \qquad \frac{}{\Gamma \vdash \mathtt{true:\ bool}}\,[\mathtt{True}] \qquad \frac{}{\Gamma \vdash \mathtt{false:\ bool}}\,[\mathtt{False}]$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e1:\ int} \qquad \Gamma \vdash \mathtt{e2:\ int} \\ \mathtt{+\ :\ int} \times \mathtt{int} \rightarrow \mathtt{int}\end{array}}{\Gamma \vdash \mathtt{e1\ +\ e2\ :\ int}}\,[\mathtt{Plus}] \qquad \frac{\Gamma(\mathtt{id}) = \mathtt{T}}{\Gamma \vdash \mathtt{id:\ T}}\,[\mathtt{Var}]$$

this is not a judgment: it is an axiom schema!

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e1:\ T1} \qquad \Gamma \vdash \mathtt{e2:\ T2} \\ \mathtt{T1{=}T2} \qquad \mathtt{==\ :\ T1} \times \mathtt{T1} \rightarrow \mathtt{bool}\end{array}}{\Gamma \vdash \mathtt{e1\ ==\ e2\ :\ bool}}\,[\mathtt{Eq}]$$

== is polymorphic!

this is the unique place where Γ is used

PROOF TREES

with the type system we can derive

$$\Gamma \vdash \texttt{(x+5)==(y+2)} : \texttt{bool}$$

assuming that $\Gamma = [\texttt{x} \mapsto \texttt{int},\ \texttt{y} \mapsto \texttt{int}]$

$$\dfrac{\text{[Plus]}\dfrac{\text{[Var]}\dfrac{\Gamma(\texttt{x}) = \texttt{int}}{\Gamma \vdash \texttt{x:int}} \qquad \dfrac{}{\Gamma \vdash \texttt{5:int}}\text{[Num]} \qquad + : \texttt{int}\times\texttt{int} \to \texttt{int}}{\Gamma \vdash \texttt{(x+5):int}} \qquad \dfrac{\dfrac{\Gamma(\texttt{y}) = \texttt{int}}{\Gamma \vdash \texttt{y:int}}\text{[Var]} \qquad \dfrac{}{\Gamma \vdash \texttt{2:int}}\text{[Num]} \qquad + : \texttt{int}\times\texttt{int} \to \texttt{int}}{\Gamma \vdash \texttt{(y+2):int}}\text{[Plus]} \qquad \texttt{==} : \texttt{int}\times\texttt{int} \to \texttt{bool}}{\Gamma \vdash \texttt{(x+5)==(y+2)}: \texttt{bool}}\text{[Eq-i]}$$

this is called **PROOF TREE**

proof trees are **finite** trees where

- the nodes are **instances** of the inference rules
- in particular, the leaves are instances of axioms
- the root of the tree contains the judgement that is demonstrated

IMPLEMENTATION OF THE ENVIRONMENT

the environments are implemented by symbol tables

- **in the following**: the type of symbol tables is `SymbolTable`
- assume Γ to be of type `SymbolTable`
- the operation of lookup in Γ is $\Gamma(\mathtt{id})$
- the operation of insert/extension of a new identifier `id` in Γ is $\Gamma[\mathtt{id} \mapsto \mathtt{type}]$

IMPLEMENTATION OF THE TYPE SYSTEM

the type checking is implemented by a recursive function on the nodes of the AST

```
Type typeCheck (SymbolTable Γ, Exp e){
   switch (e){
      case e1+e2:  if ((typeCheck(Γ,e1)==int) && (typeCheck(Γ,e2)==int))
                       return int ;
                    else { error("wrong sum") ; break ; }
      case e1==e2: if ((typeCheck(Γ,e1) == typeCheck(Γ,e2))
                       return bool ;
                   else { error("wrong equal") ; break ; }
      case id:     if (Γ(id) is undefined)
                       error("Undeclared id") ; break ;
                   else return(Γ(id)) ;
      case num:    return(int) ;
      case true:   return(bool) ;
      case false:  return(bool) ;
   }
}
```

WARNING: this is psedocode!

The `typeCheck` method in `SimpLan` has no argument because the infos are in the fields of the class

- there is no symbol table anymore… because of an optimization
- there is no case analysis because nodes are specialized

PROOFS TREES IN A TYPE CHECKING SYSTEM

with the type inference system we can derive

$$\Gamma \vdash \mathtt{(x+5)==(y+2)} : \mathtt{bool}$$

assuming that $\Gamma = [\mathtt{x} \longmapsto \mathtt{int},\ \mathtt{y} \longmapsto \mathtt{int}]$

$$
\dfrac{
\begin{array}{c}
[\mathrm{Plus}]\,\dfrac{
\begin{array}{c}
[\mathrm{Var}]\,\dfrac{\Gamma(\mathtt{x}) = \mathtt{int}}{\Gamma \vdash \mathtt{x}:\mathtt{T2}}\ \mathtt{T2=int} \quad \dfrac{}{\Gamma \vdash \mathtt{5}:\mathtt{T3}}\,[\mathrm{Num}]\ \mathtt{T3=int} \\
\mathtt{T2=int=T3} \qquad \mathtt{+} : \mathtt{int}\times\mathtt{int} \to \mathtt{int}
\end{array}
}{\Gamma \vdash \mathtt{(x+5)}:\mathtt{T1} \quad \mathtt{T1=int}}
\qquad
\dfrac{
\begin{array}{c}
\dfrac{\Gamma(\mathtt{y}) = \mathtt{int}}{\Gamma \vdash \mathtt{y}:\mathtt{T5}}\,[\mathrm{Var}]\ \mathtt{T5=int} \quad \dfrac{}{\Gamma \vdash \mathtt{2}:\mathtt{T6}}\,[\mathrm{Num}]\ \mathtt{T6=int} \\
\mathtt{T5=int=T6} \qquad \mathtt{+} : \mathtt{int}\times\mathtt{int} \to \mathtt{int}
\end{array}
}{\Gamma \vdash \mathtt{(y+2)}:\mathtt{T4} \quad \mathtt{T4=int}}\,[\mathrm{Plus}] \\
\mathtt{T1=T4} \qquad \mathtt{==} : \mathtt{T1}\times\mathtt{T1} \to \mathtt{bool}
\end{array}
}{\Gamma \vdash \mathtt{(x+5)==(y+2)} : \mathtt{T} \quad \mathtt{T = bool}}\,[\mathrm{Eq}]
$$

if you replace the type variables $\mathtt{T}, \mathtt{T1}, \ldots$ with the corresponding values, you obtain the proof tree

TYPE CHECKING OF `SimpLan`

the **`SimpLan`** language

```
prog     : exp ';'
         | let exp ';'                                                    ;
let      : 'let' (dec ';')+ 'in'                                          ;
dec      : type ID '=' exp ';'
         | type ID '(' param (',' param )* ')' '=' (let)? exp ';'         ;
type     : 'int' | 'bool'                                                 ;
exp      :  INTEGER | 'true' | 'false' | ID
         | exp '+' exp | exp '==' exp
         | 'if' '(' exp ')' '{' exp '}' 'else' '{' exp '}'
         | ID '(' exp (',' exp )* ')'                                     ;
```

`exp` are different

the following pseudocode uses explicit parameters `SymbolTable` and `Nodes` representing syntactic categories

- it is different from the implementation
- it should be simpler to understand

TYPE CHECKING OF EXPRESSIONS

we use a method `Type typeChecking(SymbolTable Γ, ExpNode e)`

the symbol table

the expression to type check

— a pointer to the syntax tree —

```
Type typeChecking(SymbolTable Γ, ExpNode e){
  switch (e){
      case num       : return(int)
      case true      :
      case false     : return(bool)
      case id        : Type t = Γ(id);
                       if (t = unbound) error("Undeclared id");
                       else return(t);
      case e1 + e2   : Type t1 = typeChecking(Γ,e1);
                        Type t2 = typeChecking(Γ,e2);
                        if ((t1==int)&&(t2==int)) return(int);
                        else error("Wrong invocation of addition");
      case e1 == e2 : Type t1 = typeChecking(Γ,e1);
                        Type t2 = typeChecking(Γ,e2);
                        if (t1 == t2) return(bool);
                        else error("Wrong invocation of conjunction");
     . . .  }
```

TYPE CHECKING OF EXPRESSIONS — CONT.

```
Type typeChecking(SymbolTable Γ, ExpNode e){
  switch (e) {
      . . .
      case if (e1){ e2 } else { e3 } :
                 Type t1 = typeChecking(Γ, e1);
                 Type t2 = typeChecking(Γ, e2);
                 Type t3 = typeChecking(Γ, e3);
                 if (t1==bool) && (t2==t3)) return(t2);
                 else error("Type mismatch in conditionals");
      id(e_list) : Type t = Γ(id) ;
                   switch (t){
                      case unbound : error("Undeclared function id");
                      case (t1,. . ., tn) -> t0 :
                        [t1’,. . ., tm’] = typeCheckingTuple(Γ, e_list)
                        if ((n==m) && (t1 == t1’) && ... && (tn == tn’))
                             return(t0);
                        else error("Wrong invocation of function");
  }
}
```

esercizio: vedere `CallNode`

```
TupleType typeCheckingTuple(SymbolTable Γ, ExpNodeList L){
 switch (L){
     case null : return([ ]);
     case e    : return([typeChecking(Γ, e)]);
     case e::L1 : return(typeChecking(Γ, e) :: typeCheckingTuple(Γ, L1));
  }}
```

element concatenation

TYPE CHECKING: ADVANCED RULES

what are the rules for conditionals and function invocations?

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e1: T1} \qquad \Gamma \vdash \mathtt{e2: T2} \qquad \Gamma \vdash \mathtt{e3: T3} \\ \mathtt{T1 = bool} \qquad \mathtt{T2 = T3}\end{array}}{\Gamma \vdash \mathtt{if\ (e1)\ e2\ else\ e3\ :\ T2}} \; [\mathtt{If}]$$

$$\frac{\begin{array}{c}\Gamma \vdash f: T_1 \times \ldots \times T_n \to T \qquad (\Gamma \vdash e_i : T_i')^{i \in 1..n} \\ (T_i = T_i')^{i \in 1..n}\end{array}}{\Gamma \vdash f(e_1, \ldots, e_n) : T} \; [\mathtt{Invk}]$$

and what about declarations?

in `SimpLan` there are two types of `declarations`

1. the declaration of an identifier `type ID '=' exp`
2. the declaration of a function `type ID '(' ( param ( ',' param)* )? ')' '=' (let)? exp`

they both change the symbol table

TYPE CHECKING OF DECLARATIONS

the judgments of `decs` are $\Gamma \vdash \texttt{decs} : \Gamma'$

the judgments return environments!

rules for declarations are

$$\frac{\Gamma \vdash \texttt{e} : \texttt{T'} \qquad \texttt{x} \notin \texttt{dom(top(}\Gamma\texttt{))} \qquad \texttt{T=T'}}{\Gamma \vdash \texttt{T x = e ;} : \Gamma[\texttt{x} \mapsto \texttt{T}]}\,[\texttt{VarD}]$$

$$\frac{\Gamma \vdash \texttt{d} : \Gamma' \qquad \Gamma' \vdash \texttt{D} : \Gamma''}{\Gamma \vdash \texttt{d D} : \Gamma''}\,[\texttt{SeqD}]$$

TYPE CHECKING OF DECLARATIONS

the rule for function declarations is

this is the rule used by `SimpLan`

it is the same if you omit the • operation!

$$\frac{\Gamma\bullet[\, x_1 \mapsto T_1,\ldots,\, x_n \mapsto T_n] \vdash e:T' \qquad T' = T \qquad f \notin \mathrm{dom}(\mathrm{top}(\Gamma))}{\Gamma \vdash T\ f(T_1\ x_1,\ldots,T_n\ x_n) = e;\ :\ \Gamma[f \mapsto (T_1,\ldots,T_n) \to T]}\ [\mathtt{Fun}]$$

`[Fun]` does not admit recursive definitions: if you want them, you must replace the judgment in the premise with

$$\Gamma[f \mapsto (T_1,\ldots,T_n) \to T]\bullet[x_1 \mapsto T_1,\ldots,\, x_n \mapsto T_n] \vdash e:T'$$

you obtain the `[FunR]` rule:

$$\frac{\Gamma[f \mapsto (T_1,\ldots,T_n) \to T]\bullet[x_1 \mapsto T_1,\ldots,\, x_n \mapsto T_n] \vdash e:T' \qquad T' = T \qquad f \notin \mathrm{dom}(\mathrm{top}(\Gamma))}{\Gamma \vdash T\ f(T_1\ x_1,\ldots,T_n\ x_n) = e;\ :\ \Gamma[f \mapsto (T_1,\ldots,T_n) \to T]}\ [\mathtt{FunR}]$$

what about mutual recursive definitions?

TYPE CHECKING OF LET

the rule for let

this corresponds to a
newScope() operation!

$$\frac{\Gamma \bullet [\,] \vdash D\!: \Gamma' \qquad \Gamma' \vdash e\!: T}{\Gamma \vdash \mathtt{let}\ D\ \mathtt{in}\ e\!: T}\ [\mathtt{Let}]$$

this corresponds to a
remove() operation!

remark: the scope of the declarations `D` are `e`: outside `let`, declarations are not accessible anymore!

question: why don't we use the simpler rule

$$\frac{\Gamma \vdash D\!: \Gamma' \qquad \Gamma' \vdash e\!: T}{\Gamma \vdash \mathtt{let}\ D\ \mathtt{in}\ e\!: T}\ [\mathtt{Let\text{-}Simpler}]$$

TYPE CHECKING: IMPLEMENTATION (WITH FUNR)

notice: it is NOT `SimpLan` because we use `[FunR]`

this may return
error("Multiple declaration of id")

```
SymbolTable typeCheckingDecs(SymbolTable Γ, DecsNode D){
        switch (D){
            case T x = e    : Type T1 = typeChecking(Γ,e) ;
                             if (T == T1) return Γ[x ⟼ T] ;
                             else error("Type mismatch in id decl") ;

            case T f(P) e   : Tuple<Types> T1 = getType(P) ;
```

// to be defined!

type checks
recursive functions
(we are using [FunR])

```
                              SymbolTable Γ' = Γ[f ⟼ T1→T] ;
                              SymbolTable Γ" = Γ'• insertArgs([],P) ;
                              Type T2 = typeChecking(Γ",e);
                              if (T == T2) return Γ' ;
                              else error("Wrong function id declaration");

    case T f(P) let D' in e: Tuple<Types> T1 = getType(P);
                              SymbolTable Γ' = Γ[f ⟼ T1→T] ;
                              SymbolTable Γ" = Γ'• insertArgs([],P) ;
                             Γ" = typeCheckingDecs(Γ",D');
                             Type T2 = typeChecking(Γ",e);
                             if (T == T2) return Γ' ;
                             else error("Wrong function id declaration");

             case d D'      : SymbolTable Γ' = typeCheckingDecs(Γ,d);
                              return typeCheckingDecs(Γ',D');
}
```

formal parameters and
local variables are in
the same nesting level
(we are using a variant of [FunR])

notice: mutual recursion is still not covered!

TYPE CHECKING FUNCTIONS: RECAP

the rules are

$$\frac{\Gamma[\mathtt{f} \longmapsto (\mathtt{T}_1,\ldots,\mathtt{T}_n) \rightarrow \mathtt{T}]\bullet[\mathtt{x}_1 \longmapsto \mathtt{T}_1,\ldots,\ \mathtt{x}_n \longmapsto \mathtt{T}_n] \vdash \mathtt{e}:\mathtt{T'} \qquad \mathtt{T'} = \mathtt{T} \quad \mathtt{f} \notin \mathtt{dom}(\mathtt{top}(\Gamma))}{\Gamma \vdash \mathtt{T}\ \mathtt{f}(\mathtt{T}_1\ \mathtt{x}_1,\ldots,\mathtt{T}_n\ \mathtt{x}_n) = \mathtt{e};\ :\ \Gamma[\mathtt{f} \longmapsto (\mathtt{T}_1,\ldots,\mathtt{T}_n) \rightarrow \mathtt{T}]}\ [\mathtt{FunR}]$$

$$\frac{\Gamma[\mathtt{f} \longmapsto (\mathtt{T}_1,\ldots,\mathtt{T}_n) \rightarrow \mathtt{T}]\bullet[\mathtt{x}_1 \longmapsto \mathtt{T}_1,\ldots,\mathtt{x}_n \longmapsto \mathtt{T}_n] \vdash \mathtt{D}:\ \Gamma[\mathtt{f} \longmapsto (\mathtt{T}_1,\ldots,\mathtt{T}_n) \rightarrow \mathtt{T}]\bullet\Gamma' \qquad \Gamma[\mathtt{f} \longmapsto (\mathtt{T}_1,\ldots,\mathtt{T}_n) \rightarrow \mathtt{T}]\bullet\ \Gamma' \vdash \mathtt{e}:\mathtt{T'} \qquad \mathtt{T'} = \mathtt{T} \quad \mathtt{f} \notin \mathtt{dom}(\mathtt{top}(\Gamma))}{\Gamma \vdash \mathtt{T}\ \mathtt{f}(\mathtt{T}_1\ \mathtt{x}_1,\ldots,\mathtt{T}_n\ \mathtt{x}_n) = \mathtt{let}\ \mathtt{D}\ \mathtt{in}\ \mathtt{e};\ :\ \Gamma[\mathtt{f} \longmapsto (\mathtt{T}_1,\ldots,\mathtt{T}_n) \rightarrow \mathtt{T}]}\ [\mathtt{FunRLet}]$$

notice: in `SimpLan` we use `[FunR]`, NOT `[FunR]`

TYPE CHECKING PROGRAMS

programs are

```
    prog    : 'let' (vardec | fundec)+ 'in' exp ';'      ;
```

then the inference rule is the same of `let` and the implementation is

```
Type typeChecking(SymbolTable Γ, ProgNode p){// the symbol table is not used!
        SymbolTable Γ'= typeCheckingDecs(∅, p.decs);
        return typeChecking(Γ', p.exp) ;
}
```

problem: how to let a function invokes another function defined afterwards?

TYPE CHECKING PROGRAMS (MUTUAL RECURSION)

we need a pre-visit that collects function definitions

formally, there is a new judgment $\Gamma \Vdash \mathtt{D} : \Gamma'$

$$\frac{}{\Gamma \Vdash \mathtt{T\ x = e\ ;} : \Gamma}\ \mathtt{[VarM]} \qquad \frac{\Gamma \Vdash \mathtt{d} : \Gamma' \qquad \Gamma' \Vdash \mathtt{D} : \Gamma''}{\Gamma \Vdash \mathtt{d\ D} : \Gamma''}\ \mathtt{[DecM]}$$

$$\frac{\mathtt{f} \notin \mathtt{dom(top(}\Gamma\mathtt{))}}{\Gamma \Vdash \mathtt{T\ f(T_1\ x_1,\ldots,T_n\ x_n) = e;} : \Gamma[\mathtt{f} \mapsto (\mathtt{T_1},\ldots,\mathtt{T_n}) \rightarrow \mathtt{T}]}\ \mathtt{[FunM]}$$

the `let`-rule becomes

here we use [Fun] and not [FunR].
WHY? WHAT HAPPENS IF WE USE [FunR]?

$$\frac{\varnothing \Vdash \mathtt{D} : \Gamma' \qquad \Gamma \bullet \Gamma' \bullet [\,] \vdash \mathtt{D} : \Gamma'' \qquad \Gamma'' \vdash \mathtt{e} : \mathtt{T}}{\Gamma \vdash \mathtt{let\ D\ in\ e} : \mathtt{T}}\ \mathtt{[LetM]}$$

question: `[LetM]` has been written with the premise $\Gamma \bullet \Gamma' \bullet [\,] \vdash \mathtt{D} : \Gamma''$ for reusing `[Fun]` (or `[FunR]`): may you provide a different rule in order to have the simpler premise $\Gamma \bullet \Gamma' \vdash \mathtt{D} : \Gamma''$?

TYPE CHECKING PROGRAMS (MUTUAL RECURSION)

```
SymbolTable typeCheckingDecsAux(SymbolTable Γ, DecsNode D){
      case D of
               Empty   : return(Γ);
         T x = e ; D' : return typeCheckingDecsAux(Γ, D') ;
        T f(A) B ; D' : Tuple<Types> T' = getType(A) ;
                        SymbolTable Γ' = insert(Γ, f, T'->T) ;
                        return typeCheckingDecsAux(Γ, D') ;
}
```

the insert may also fail:
multiple declarations of functions in the same scope

the type inference of the whole program managing mutual recursion is

```
SymbolTable typeCheckingDecs(SymbolTable Γ, ProgNode p){
     SymbolTable Γ' = typeCheckingDecsAux(EMPTY_TABLE, p.decs);
     Γ' = newScope(Γ') ;
     SymbolTable Γ" = typeCheckingDecs(Γ', p.decs);
     return typeChecking(Γ", p.exp) ;
 }
```

ADVANCED ISSUES

- subtyping
- statements
- subtyping and assignment
- overriding

SUBTYPING

consider the following program in `miniSimpLan`:

```
let T x = e in e′
```

according to the current type system we have the proof tree

$$\dfrac{\text{[VarD]}\dfrac{\Gamma\bullet[\,]\vdash \texttt{e}:\texttt{T"} \qquad \texttt{x}\notin \texttt{dom}(\texttt{top}(\Gamma\bullet[\,])) \qquad \texttt{T}=\texttt{T"}}{\Gamma\bullet[\,]\vdash \texttt{T x = e}:\ \Gamma\bullet[\texttt{x}\mapsto \texttt{T}]} \qquad \Gamma\bullet[\texttt{x}\mapsto\texttt{T}]\vdash \texttt{e}':\texttt{T}'}{\Gamma\vdash \texttt{let T x = e in e}':\texttt{T}'}\text{[Prog]}$$

the equality between `T"` and `T` is a constraint that is sometimes **too strong**

example: with the above type system it is not possible to type

```
class C inherits P { ... }
            ...
let P x = new C in ...
```

problems with **inheritance**!

SUBTYPING

define a relation `T <: T'` on types to say that:

- an object of type `T` could be used when one of type `T'` is **acceptable**

Definition: subtyping

Let `Inherits_from` be a set of pairs of types. A relation `<:` on types is called **subtyping** when

- `T <: T`
- `T <: T'` if $(T,T') \in$ `Inherits_from`
- `T <: T'` if `T <: T"` and `T" <: T'`

`VarD` with subtyping:

$$\frac{\Gamma \vdash e : T \qquad x \notin \mathrm{dom}(\mathrm{top}(\Gamma)) \qquad T <: T'}{\Gamma \vdash T'\ x = e\ ;\ : \Gamma[x \mapsto T']}\ [\texttt{Var-Subt}]$$

the old rule was:

$$\frac{\Gamma \vdash e : T' \qquad x \notin \mathrm{dom}(\mathrm{top}(\Gamma)) \qquad T=T'}{\Gamma \vdash T\ x = e\ ;\ : \Gamma[x \mapsto T]}\ [\texttt{VarD}]$$

BE CAREFUL: WRONG DEC/LET RULE

when declarations are singletons you may have a compact dec-let rule:

$$\frac{\begin{array}{c}\Gamma \vdash e : T \qquad \Gamma[x \longmapsto T'] \vdash e' : T'' \\ T <: T'\end{array}}{\Gamma \vdash \mathtt{let}\ T'\ x = e\ \mathtt{in}\ e' : T''}\ [\mathtt{CompactLet}]$$

no need to have •

WHY?

1. consider the following wrong rule:

$$\frac{\begin{array}{c}\Gamma \vdash e : T \qquad \Gamma \vdash e' : T'' \\ T <: T'\end{array}}{\Gamma \vdash \mathtt{let}\ T'\ x = e\ \mathtt{in}\ e' : T''}\ [\mathtt{WrongLet1}]$$

- the following good program does not typecheck

```
let int x = 0 in x + 1
```

- and some bad programs do typecheck

```
int foo(B x){ let A x = new A in x.b() }
```

[the problem was that `e'` was typed in a wrong env]

BE CAREFUL: WRONG DEC/LET RULE

2. next, consider another hypothetical dec/let rule:

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \qquad \Gamma[\mathtt{x} \longmapsto \mathtt{T'}] \vdash \mathtt{e'} : \mathtt{T''} \qquad \mathtt{T'} <: \mathtt{T}}{\Gamma \vdash \mathtt{let}\ \mathtt{T'}\ \mathtt{x} = \mathtt{e}\ \mathtt{in}\ \mathtt{e'} : \mathtt{T''}}\ [\mathtt{WrongLet2}]$$

- the following bad program is well typed

```
let B x = new A in x.b()
```

[the problem is that we have inverted the subtyping relation]

3. then consider this dec/let rule:

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \qquad \Gamma[\mathtt{x} \longmapsto \mathtt{T}] \vdash \mathtt{e'} : \mathtt{T''} \qquad \mathtt{T} <: \mathtt{T'}}{\Gamma \vdash \mathtt{let}\ \mathtt{T'}\ \mathtt{x} = \mathtt{e}\ \mathtt{in}\ \mathtt{e'} : \mathtt{T''}}\ [\mathtt{WrongLet3}]$$

- the following good program is not typed

```
let A x = new B in {... x = new A; x.a(); }
```

[the problem is that `e'` has been typed with a wrong binding for `x`]

FUNCTION INVOCATION WITH SUBTYPING

function `f` with type: $T_1 \times ... \times T_n \rightarrow T$

$$\frac{\Gamma \vdash f : T_1 \times \ldots \times T_n \rightarrow T \qquad (\Gamma \vdash e_i : T_i')^{i \in 1..n} \qquad (T_i' <: T_i)^{i \in 1..n}}{\Gamma \vdash f(e_1, ..., e_n) : T}\ [\text{Invk-Subt}]$$

- therefore a function may be invoked with values of a subtype

CONDITIONAL WITH SUBTYPING

the syntax of conditional expressions:

`if (e) { e1 } else { e2 }`

- the typing system infer types that may be all different …

$$\frac{\begin{array}{ccc}\Gamma \vdash \texttt{e: T} & \Gamma \vdash \texttt{e1: T1} & \Gamma \vdash \texttt{e2: T2} \\ \texttt{T <: bool} & \texttt{T1 <: T'} & \texttt{T2 <: T'}\end{array}}{\Gamma \vdash \texttt{if (e) \{ e1 \} else \{ e2 \} : T'}}\;\texttt{[If-Subt]}$$

STATEMENTS AND TYPE CHECKING

extend `miniSimpLan` with statements

- you get `impSimpLan`

sequences of statements

```
prog    : 'let' decs 'in' ( exp | stats )  ';'                     ;
decs    : ( vardec | fundec )+                                     ;
vardec  : type ID '=' exp ';'                                      ;
fundec  : type ID '(' ( args )? ')' fbody ';'                      ;
fbody   : exp | stats | 'let' (vardec)+ 'in' ( exp | stats )       ;
args    : type ID ( ',' type ID)*                                  ;
type    : 'int' | 'bool' | 'void'                                  ;
exp     :  INTEGER | 'true' | 'false' | ID
        | exp '+' exp | exp '==' exp
        | 'if' exp 'then' '{' exp '}' 'else' '{' exp '}'
        | ID '(' (exps)? ')'                                       ;
exps    : exp (',' exps )*                                         ;
stats   : stat (';' stat )*                                        ;
stat    : ID ':=' exp
        | 'if' exp '{' stats '}' 'else' '{' stats '}'              ;
```

bodies may also be statements

the type void!

the statements

THE TYPING SYSTEM FOR STATEMENTS

$$\frac{\begin{array}{c}\Gamma(x) = T \qquad \Gamma \vdash e\!: T' \\ T = T'\end{array}}{\Gamma \vdash x = e; \ : \mathtt{void}}\ \text{[Asgn]}$$

$$\frac{\begin{array}{c}\Gamma \vdash e\!: T \qquad \Gamma \vdash s1\!: T' \qquad \Gamma \vdash s2\!: T'' \\ T = \mathtt{bool} \qquad T' = \mathtt{void} = T''\end{array}}{\Gamma \vdash \mathtt{if}\ (e)\ \{\ s1\ \}\ \mathtt{else}\ \{\ s2\ \}\ : \mathtt{void}}\ \text{[IfS]}$$

$$\frac{\begin{array}{c}\Gamma \vdash s\!: T \qquad \Gamma \vdash S : T' \\ T = \mathtt{void} = T'\end{array}}{\Gamma \vdash s\ S\ : \mathtt{void}}\ \text{[SeqS]}$$

- not so difficult

THE TYPING SYSTEM WITH SUBTYPING FOR STATEMENTS

assignment with subtyping!

$$\frac{\begin{array}{c}\Gamma(x) = T \qquad \Gamma \vdash e : T' \\ T' <: T\end{array}}{\Gamma \vdash x = e\ ;\ : \mathtt{void}}\ [\mathtt{Asgn\text{-}Subt}]$$

T' <: void means that T1 = void because there is no subtype of void

$$\frac{\begin{array}{ccc}\Gamma \vdash e : T & \Gamma \vdash s1 : T' & \Gamma \vdash s2 : T'' \\ T <: \mathtt{bool} & T' <: \mathtt{void} & T'' <: \mathtt{void}\end{array}}{\Gamma \vdash \mathtt{if}\ (e)\ s1\ \mathtt{else}\ s2 : \mathtt{void}}\ [\mathtt{IfS\text{-}Subt}]$$

$$\frac{\begin{array}{cc}\Gamma \vdash s : T & \Gamma \vdash S : T' \\ T <: \mathtt{void} & T' <: \mathtt{void}\end{array}}{\Gamma \vdash s\ S : \mathtt{void}}\ [\mathtt{SeqS\text{-}Subt}]$$

- therefore, if `B <: A` , we may write

```
A x = new B() ;
```

COVARIANT ARRAYS AND ASSIGNMENTS

let `array[A]` be the type of an array containing data of type `A`

one could assume:

- `if B <: A then array[B] <: array[A]`

  (known as **covariant arrays**, present e.g. in **`Java`**)

but, consider the following program (well typed according to this assumption):

```
let void f(x:array[A]){ x[1]= new A; }
in let z : array[B]
   in { f(z); z[1].b(); };
```

we are assigning to a variable of type B a value of type A with B <: A

what is wrong with this program?

A (GENERALLY) WRONG SUBTYPING RULE (CONT.)

problem:

- when the **array of subtypes** is used in place of an **array of supertypes**…
- …it is possible to **insert a supertype** in the array…
- …and then the supertype can be used in place of a subtype (type error!)

but if arrays **cannot be written/modified**, "covariance" is sound!

- it is admitted when the array items **are NOT assigned**

CLASS TYPING RULE

what is the type of:

$$\begin{array}{l}\texttt{class A\{}\\ \quad T_1\ f_1;\ \ldots\ T_n\ f_n;\\ \quad T_1'\ m_1(\overline{T_1''\ x_1})\{\ldots\}\\ \quad \ldots\\ \quad T_h'\ m_h(\overline{T_h''\ x_h})\{\ldots\}\\ \texttt{\}}\end{array}$$

$$D = T_1\ f_1;\ \ldots\ T_n\ f_n;\ T_1'\ m_1(\overline{T_1''\ x_1})\{\ldots\}\ \ldots\ T_h'\ m_h(\overline{T_h''\ x_h})\{\ldots\}$$

$$\frac{\Gamma[A \longmapsto [f_i \longmapsto T_i\ , m_j \longmapsto \overline{T_j''} \rightarrow T_j']^{i\in 1..n, j\in 1..h}\ ]\ \vdash\ D:\ \Gamma'}{\Gamma\ \vdash\ \texttt{class A\{...\}}:\ \Gamma[A \longmapsto [f_i \longmapsto T_i\ , m_j \longmapsto \overline{T_j''} \rightarrow T_j']^{i\in 1..n, j\in 1..n}\ ]}\ [\texttt{Class}]$$

CLASS SUBTYPING

subclasses can usually override some declarations of the superclass

- usually the body assigned to method

assume it is possible to **override both fields and methods**, by changing also their types

problem:

```
class A{..., T f, ... }
 class B inherits A {..., T’ f, ... }
```

with `T’ <: T` (fields can be seen as array cells)

field overriding is usually not admitted

- if they can be dynamically modified, the type cannot be changed
- in a **functional language** (e.g. `SimpLan`) field subtyping may be supported (**covariant fields**)

METHOD INVOCATION

this is a small environment as well!

remark: the environment Γ now binds class types

- e.g. $\Gamma(\texttt{C}) = [\texttt{a} : \texttt{T}_\texttt{a},\ \texttt{b} : \texttt{T}_\texttt{b},\ \texttt{m} : \texttt{T}_1 \times \ldots \times \texttt{T}_n \to \texttt{T}]$
- therefore we may access to the type of a field with $\Gamma(\texttt{C})(\texttt{a})$ and of a method with $\Gamma(\texttt{C})(\texttt{m})$
- you may also use the notations $\Gamma(\texttt{C.a})$ and $\Gamma(\texttt{C.m})$

the type rule for method invocation:

$$\frac{\Gamma(\texttt{x}) = \texttt{C} \qquad \Gamma(\texttt{C.m}) = \texttt{T}_1 \times \ldots \times \texttt{T}_n \to \texttt{T} \qquad (\Gamma \vdash \texttt{e}_i : \texttt{T}_i' \qquad \texttt{T}_i' <: \texttt{T}_i)^{i \in 1..n}}{\Gamma \vdash \texttt{x.m}(\texttt{e}_1, \ldots, \texttt{e}_n) : \texttt{T}} \ \texttt{[MtdInvk-Subt]}$$

METHOD OVERRIDING (IN SCALA)

example [from `Scala`]:

$$\texttt{class A\{..., T m(T}_1\texttt{ p}_1\texttt{,..., T}_n\texttt{ p}_n\texttt{)\{ e \},... \}}$$

$$\texttt{class B inherits A\{..., T' m(T}_1\texttt{' p}_1\texttt{,..., T}_n\texttt{' p}_n\texttt{)\{ e' \}, ... \}}$$

consider $\texttt{let T z = x.m(e}_1\texttt{,...,e}_n\texttt{) in e}$

assuming `x` with type `A` that can be instantiated by an object of type `B`

- the `B` return type `T'` must be usable in place of the `A` return type `T`
  - we need `T' <: T`
- `A` parameter types must be usable in place of **`B`** parameter types
  - we need $\texttt{T}_\texttt{i}\texttt{ <: T}_\texttt{i}\texttt{'}$

METHOD OVERRIDING: SUMMARY

- if $T_1 <: T_1' \;.\;.\;.\; T_n <: T_n'$ and $T' <: T$ and $B <: A$
- the type of $\mathtt{m}$ in $\mathtt{A}$ is $T_1 \times \ldots \times T_n \rightarrow T$
- the type of $\mathtt{m}$ in $\mathtt{B}$ is $T_1' \times \ldots \times T_n' \rightarrow T'$
- then

controvariance covariance

$$\frac{(T_i <: T_i')^{i \in 1..n} \qquad T' <: T}{T_1' \times \ldots \times T_n' \rightarrow T' <: T_1 \times \ldots \times T_n \rightarrow T}\ \texttt{[Arrow-Subt]}$$

- which is the general subtyping rule for functions
  - **covariant** output (return) type
  - **contravariant** input (parameter) types
  - most of the programming languages (**`Java`**, **`C#`**) only admit **invariance** of input types!

APPENDIX

snippets of type checking in `SimpLan`

SNIPPETS OF TYPE CHECKING EXPRESSIONS IN SIMPLAN

this is in `PlusNode.java`

```
public Type typeCheck() {
   if ((left.typeCheck() instanceof IntType) && (right.typeCheck() instanceof IntType))
       return new IntType() ;
   else {
       System.out.println("Type Error: Non integers in addition") ;
       return new ErrorType() ;
   }
}
```

returns INT

this is in `EqualNode.java`

```
public Type typeCheck() {
   Type tl = left.typeCheck() ;
   Type tr = right.typeCheck();
   if (tl.getClass().equals(tr.getClass()))
       return new BoolType() ;
   else {
       System.out.println("Type Error: Different types in equality") ;
       return new ErrorType() ;
   }
}
```

returns BOOL

SNIPPETS OF TYPE CHECKING CONDITIONALS IN SIMPLAN

this is in `IfNode.java`

$$\frac{\Gamma \vdash \mathtt{e1{:}\ T1} \qquad \Gamma \vdash \mathtt{e2{:}\ T2} \qquad \Gamma \vdash \mathtt{e3{:}\ T3} \qquad \mathtt{T1 = bool} \qquad \mathtt{T2 = T3}}{\Gamma \vdash \mathtt{if\ (e1)\ e2\ else\ e3\ :\ T2}}\ \mathtt{[If]}$$

```
public Type typeCheck() {
    if (guard.typeCheck() instanceof BoolType) {
        Type thenexp = thenbranch.typeCheck() ;
        Type elseexp = elsebranch.typeCheck() ;
        if (thenexp.getClass().equals(elseexp.getClass()))
            return thenexp;
        else {
            System.out.println("Type Error: different types in then and else");
            return new ErrorType() ;
        }
    } else {
        System.out.println("Type Error: non boolean condition in if");
        return new ErrorType() ;
    }
}
```

SNIPPETS OF SIMPLAN

$$\frac{\Gamma\bullet[\ x_1 \mapsto T_1,\dots,\ x_n \mapsto T_n] \vdash e:T' \qquad T' = T \qquad f \notin dom(top(\Gamma))}{\Gamma \vdash T\ f(T_1\ x_1,\dots,T_n\ x_n) = e;\ :\ \Gamma[f \mapsto (T_1,\dots,T_n) \to T]}\text{[Fun]}$$

in `FunNode.java:`

```
public Type typeCheck () {
     if (declist!=null)
          for (Node dec:declist)
               dec.typeCheck();
     if ( (body.typeCheck()).getClass().equals(returntype.getClass()))
          return null ;
     else {
          System.out.println("Wrong return type for function "+id);
          return new ErrorType() ;
     }
}
```

$$\frac{\Gamma \vdash e : T' \qquad x \notin dom(top(\Gamma)) \qquad T=T'}{\Gamma \vdash T\ x = e\ ;\ :\ \Gamma[x \mapsto T]}\text{[VarD]}$$

in `DecNode.java:`

```
public Type typeCheck () {
     if (type.gettype() instanceof ArrowType) {
          System.out.println("Wrong usage of function identifier");
          return new ErrorType() ;
     } else return type.gettype() ;
}
```

FINAL COMMENTS

- the typing rules use very concise notation
- they are very carefully constructed
- but some **good programs** will be **rejected** anyway

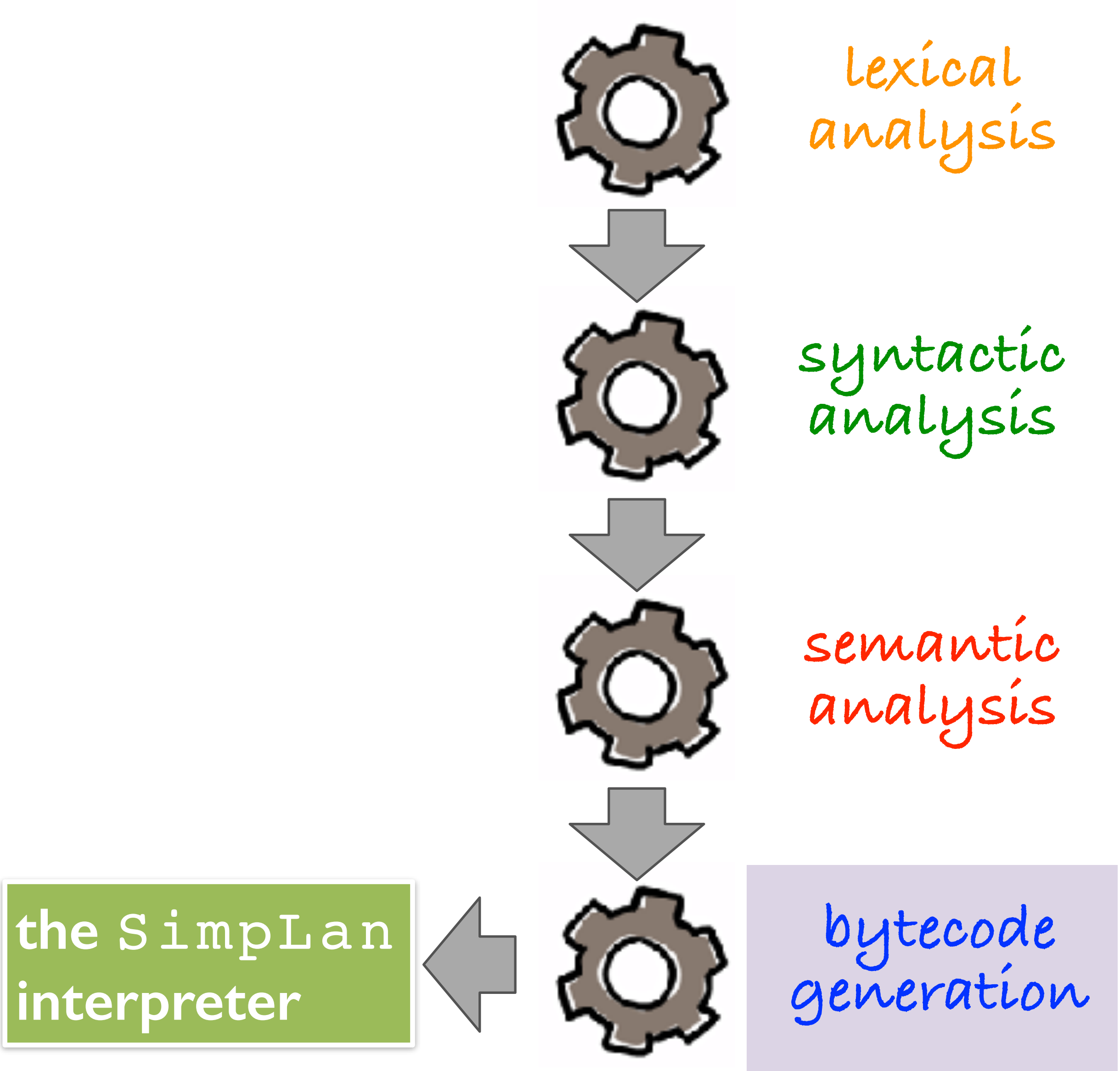
```
if (x == x) then return(x==1) else return(x)
```

Rice Theorem **the notion of good program is undecidable**

a type system enables a compiler to detect many common programming errors

- the cost is that some correct programs are disallowed
- one might have more expressive static type checking
- but more expressive type systems are also more complex

NEXT LECTURE



lexical analysis

syntactic analysis

semantic analysis

bytecode generation

the SimpLan interpreter