Type checking means looking at a parsed program to make sure that:

- every piece of the program is well-typed;
- all identifiers are used in a type-consistent way.

Input:
- program AST or parse tree (or perhaps done during parsing)

Outputs:
- “OK” or typing error message(s)
- Perhaps type-annotated AST, perhaps per-identifier type information, etc.

Will see how to specify type-checking rules using
- Attribute grammars
- Inference rules
To type-check code that uses variables, function names, or other identifiers, must maintain an environment mapping identifiers to types.

- Environment is just a dictionary with key=identifier and value=type.
- Many possible representations: hash table, linked list or tree structure, etc.
- Current environment grows and shrinks as identifiers enter and leave scope.

It is often convenient to treat the current environment as a parameter of the type-checking code, so it can vary as we do a recursive traversal.

A more traditional approach is to treat the environment as a global whose contents are mutated as we traverse the program. In this case, the environment is usually called a symbol table.
Initially, we will show how to specify type-checking using attribute grammars over parse trees.

- Calculate a synthesized type attribute for each expression.
- Check that expressions are used correctly within other expressions and statements.
- Maintain environment information in an attribute or a global symbol table.
Motivating Inherited Attributes

Sometimes it’s convenient to make a node’s attributes dependent on siblings or ancestors in tree.

Useful for expressing dependence on context, e.g., relating identifier uses to declarations. (This is especially important because CF grammar cannot capture such dependencies.)

Example: Simple C-like Variable Declarations

\[ D \rightarrow T \; L \]
\[ T \rightarrow \text{int} \mid \text{real} \]
\[ L \rightarrow L_1, \text{id} \mid \text{id} \]

Parse tree for \text{real} \, a, b, c:
INHERITED ATTRIBUTE GRAMMAR

\[
\begin{align*}
D & \rightarrow TL \\
T & \rightarrow \text{int} \quad T\.type \ := \ integer \\
T & \rightarrow \text{real} \quad T\.type \ := \ real \\
L & \rightarrow L_1, \text{id} \quad \{ \quad L_1\.type \ := \ L\.type; \ \text{addsym} \text{b} \} \\
L & \rightarrow \text{id} \quad \text{addsym} \text{b}
\end{align*}
\]

Here \text{addsym} adds \text{id} and its type to symbol table, and \text{T\.type} is an inherited attribute.

A parse tree showing dependency relations among attributes:

\[
\begin{align*}
D & \\
\rightarrow & \\
T\.type & := \text{real} \\
\rightarrow & \\
L\.type & := \text{real} \\
\rightarrow & \\
\quad & , \quad \text{id}.name \ := \ c \\
\rightarrow & \\
L\.type & := \text{real} \quad , \quad \text{id}.name \ := \ b \\
\rightarrow & \\
\quad & \text{id}.name \ := \ a
\end{align*}
\]
Dependency arrows for a dependency graph; we must evaluate attributes in **topological** order of dependency graph.

If attributes are defined on parse tree, may want to evaluate attributes while (or instead of) building the tree. This is **sometimes** possible:

- Saw how to evaluate **S-attributed** grammar, in which all attributes are synthesized, during bottom-up parsing; this method doesn’t work for inherited attributes.

- Top-down parser can easily evaluate **L-attributed** grammars, in which attributes don’t depend on their right ancestors. (Bottom-up parsers can sometimes handle these too, though with difficulty.) Example follows.

- For more complicated attribute grammars, might have to build some or all of tree **before** evaluating attributes.
Attribute Evaluation during Recursive Descent

Each non-terminal function now takes \textit{inherited} attribute values as \textit{arguments} and return (record of) \textit{synthesized} attribute value(s) as result.

Example revisited (with left-recursion removed):

```java
class Ty {}
static Ty intTy = new Ty(); static Ty realTy = new Ty();

void D() { Ty ty = T(); L(ty); }
Ty T() {
    if (tok == INT) {
        tok = lex(); return intTy;
    } else if (tok == REAL) {
        tok = lex(); return realTy;
    } else error(); }
void L(Ty ty) {
    if (tok == ID) {
        addsymb(lexeme,ty); tok = lex();
    } else error();
    if (tok == ',') {
        tok = lex(); L(ty);}
}
```
When using bottom-up parser (e.g., with \texttt{yacc} or \texttt{CUP}), it is desirable to avoid inherited attributes.

There are several approaches:

- Move the activity requiring the attribute to a higher node in the tree, by substituting a synthesized attribute for the inherited one, e.g.:

\[
\begin{align*}
D \rightarrow T \; L & \quad \text{for each $id$ in $L$.list} \\
& \quad \text{addsym} (id.name, T.type) \\
T \rightarrow \text{int} & \quad T.type := \text{integer} \\
T \rightarrow \text{real} & \quad T.type := \text{real} \\
L \rightarrow L_1, id & \quad L.list := \text{append-list(id,L}_1\.list) \\
L \rightarrow id & \quad L.list := \text{singleton-list(id)}
\end{align*}
\]
Avoiding Inherited Attributes (2)

- Can sometimes rewrite grammar, e.g.:

\[
\begin{align*}
D \rightarrow T \text{id} & \quad \{ D\.\text{type} := T\.\text{type}; \notag \\
& \quad \text{addsym}(\text{id}.\text{name}, T\.\text{type}) \} \\
D \rightarrow D_1, \text{id} & \quad \{ D\.\text{type} := D_1\.\text{type}; \notag \\
& \quad \text{addsym}(\text{id}.\text{name}, D\.\text{type}) \} \\
T \rightarrow \text{int} & \quad T\.\text{type} := \text{integer} \\
T \rightarrow \text{real} & \quad T\.\text{type} := \text{real}
\end{align*}
\]
Attribute grammar method extends to **abstract** grammars (not intended for parsing), e.g., AST grammars.

- Same concept, but attribute evaluation always occurs after whole tree is built.
- Can use recursive descent as an attribute evaluation technique (regardless of how parsing was performed).
- Typical applications: typechecking, code generation, interpretation.

Why attribute grammars?

- **Compact**, convenient formalism.
- **Local** rules describe entire computation.
- Separate **traversal** from **computation**.
- (Purely **functional** rules can be evaluated in any order.)
Can view checking process as evaluation of following attribute grammar, where

- $\text{exp.ok}$ and $\text{exp.ok}$ are synthesized boolean attributes indicating whether expression has checked successfully; and

- $\text{exp.env}$ and $\text{exp.env}$ are inherited environment attributes (with operators $\text{empty}$, $\text{extend}$, and $\text{lookup}$) containing entries for all in-scope variables.
\[
\begin{align*}
\text{program} & \rightarrow \text{exp} & \text{exp.env} & := \text{empty} \\
\text{exp} & \rightarrow \text{ID} & \text{exp.ok} & := \text{lookup(exp.env,ID.name)} \\
& \quad \rightarrow \text{NUM} & \text{exp.ok} & := \text{true} \\
& \quad \rightarrow \text{exp}_1 \ '+' \text{exp}_2 & \{ \text{exp}_1.env := \text{exp}_2.env = \text{exp.env}; \\
& & \quad \text{exp.ok} := \text{exp}_1.ok \text{ AND } \text{exp}_2.ok \} \\
& \quad \rightarrow \text{exp}_1 \ '-' \text{exp}_2 & \{ \text{exp}_1.env := \text{exp}_2.env = \text{exp.env}; \\
& & \quad \text{exp.ok} := \text{exp}_1.ok \text{ AND } \text{exp}_2.ok \} \\
& \quad \rightarrow \text{ID} \ '=' \text{exp}_1 & \{ \text{exp}_1.env := \text{exp.env}; \\
& & \quad \text{exp.ok} := \text{lookup(exp.env,ID.name)} \text{ AND } \text{exp}_1.ok \} \\
& \quad \rightarrow \text{if0} \text{ exp}_1 \text{ exp}_2 \text{ exp}_3 & \{ \text{exp}_1.env := \text{exp}_2.env := \text{exp}_3.env := \text{exp.env}; \\
& & \quad \text{exp.ok} := \text{exp}_1.ok \text{ AND } \text{exp}_2.ok \text{ AND } \text{exp}_3.ok \} \\
& \rightarrow \text{'} \text{ vars } \text{'} \text{;'} \text{ exps } \text{'} \} & \{ \text{exps.env} := \text{extend(exp.env,vars)}; \\
& & \quad \text{exp.ok} := \text{exps.ok} \} \\
\text{exps} & \rightarrow \text{exp} \{ \text{exp.env} := \text{exps.env}; \\
& & \quad \text{exps.ok} := \text{exp.ok} \} \\
& \rightarrow \text{exp} \text{'};\text{'} \text{ exps}_1 & \{ \text{exp.env} := \text{exps}_1.env := \text{exps.env}; \\
& & \quad \text{exps.ok} := \text{exp.ok} \text{ AND } \text{exps}_1.ok \} \\
\end{align*}
\]
Consider a simple language of declarations, statements, and expressions.

\[ P \rightarrow D ; S \quad \{ \quad S.env = D.env; \quad \} \]

Actions for declarations synthesize environment attributes:

\[ D \rightarrow \epsilon \quad \{ \quad D.env := \text{empty} \quad \} \]
\[ D \rightarrow \text{id} : T_1 ; D_1 \quad \{ \quad D.env := \text{extend}(D_1.env, \text{binding}(\text{id}, T_1.type)) \quad \} \]
\[ T \rightarrow \text{bool} \quad \{ \quad T.type := \text{boolean} \quad \} \]
\[ T \rightarrow \text{int} \quad \{ \quad T.type := \text{integer} \quad \} \]
\[ T \rightarrow \text{array of} \ T_1 \quad \{ \quad T.type := \text{array}(T_1.type) \quad \} \]
\[ T \rightarrow \text{pair} \ T_1 \ T_2 \quad \{ \quad T.type := T_1.type \times T_2.type \quad \} \]
Actions for expressions **check** for compatible operands and **synthesize** attribute type:

\[
\begin{align*}
E \rightarrow &\text{num} \quad \{ \ E.type := \text{integer} \ \} \\
E \rightarrow &\text{id} \quad \{ \ E.type := \text{lookup}(E.env, id) \ \} \\
E \rightarrow (E_1, E_2) \quad \{ \ E_1.env = E.env; \ E_2.env = E.env; \ E.type = E_1.type \times E_2.type \ \} \\
E \rightarrow &E_1 \div E_2 \quad \{ \ E_1.env = E.env; \ E_2.env = E.env; \\
&\quad \text{if not } (E_1.type = \text{integer} \text{ and } E_2.type = \text{integer}) \text{ then} \\
&\quad \text{issue type error;} \\
&\quad E.type := \text{integer} \} \\
E \rightarrow &E_1 \text{ or } E_2 \quad \{ \ E_1.env = E.env; \ E_2.env = E.env; \\
&\quad \text{if not } (E_1.type = \text{boolean} \text{ and } E_2.type = \text{boolean}) \text{ then} \\
&\quad \text{issue type error;} \\
&\quad E.type := \text{boolean} \}
\end{align*}
\]

Issuing error might or might not stop the checking process. If it doesn’t, try to choose a synthesized type value that prevents a cascade of messages from a single mistake.
MORE EXPRESSIONS

\[ E \rightarrow E_1 \ [ \ E_2 \ ] \quad \{ \ E_1.env = E.env; \ E_2.env = E.env; \]
\[ \text{if } (E_1.type = \text{array}(T) \text{ and } E_2.type = \text{integer}) \text{ then} \]
\[ \quad E.type := T; \]
\[ \text{else issue type error} \} \]

\[ E \rightarrow E_1.fst \quad \{ \ E_1.env = E.env; \]
\[ \quad \text{if } E_1 = T_1 \times T_2 \text{ then} \]
\[ \quad \quad E.type := T_1; \]
\[ \quad \text{else issue type error;} \}

\[ E \rightarrow E_1 < E_2 \quad \{ \ E_1.env = E.env; \ E_2.env = E.env; \]
\[ \quad \text{if not } (E_1.type = \text{integer and } E_2.type = \text{integer}) \text{ then} \]
\[ \quad \quad \text{issue type error;} \]
\[ \quad \quad E.type := \text{boolean} \} \]

\[ E \rightarrow E_1 = E_2 \quad \{ \ E_1.env = E.env; \ E_2.env = E.env; \]
\[ \quad \text{if not } ((E_1.type = \text{boolean or } E_1.type = \text{integer}) \]
\[ \quad \quad \text{and } E_1.type = E_2.type) \text{ then} \]
\[ \quad \quad \text{issue type error;} \]
\[ \quad \quad E.type := \text{boolean} \} \]
In most languages, statements don’t have a type, so no point in synthesizing an attribute. Actions just check component types:

$$S \rightarrow \text{id} := E_1 \quad \{ \text{E}_1.\text{env} = S.\text{env}; \text{if E}_1.\text{type} \neq \text{lookup}(S.\text{env}, \text{id}) \text{ then} \text{issue type error} \}$$

(Must also check that id is an l-value that can be assigned into.)

$$S \rightarrow \text{if } E_1 \text{ then } S_1 \quad \{ \text{E}_1.\text{env} = S.\text{env}; S_1.\text{env} = S.\text{env}; \text{if E}_1.\text{type} \neq \text{boolean then} \text{issue type error} \}$$

$$S \rightarrow S_1 ; S_2 \quad \{ \text{S}_1.\text{env} = S.\text{env}; \text{S}_2.\text{env} = S.\text{env}; \}$$
Can describe type of function as $\text{type}_1 \times \text{type}_2 \times \ldots \times \text{type}_n \rightarrow \text{type}$

\[
\begin{align*}
D \rightarrow \text{id} ( F_1 ) & : T_1 ; D_1 & \{ \text{D.env := extend(D}_1\text{.env, binding(id,F}_1\text{.type} \rightarrow T_1\text{.type})} \} \\
F \rightarrow \text{id} & : T_1 & \{ \text{F.type := T}_1\text{.type} \} \\
F \rightarrow \text{id} , F_1 & : T_1 & \{ \text{F.type := T}_1\text{.type} \times F_1\text{.type} \}
\end{align*}
\]

\[
\begin{align*}
E \rightarrow \text{id} ( A_1 ) & \{ \text{A}_1\text{.env = E.env;} \\
& \quad \text{if lookup(E.env, id) = T}_1 \rightarrow T_2 \text{ then} \\
& \quad \quad \text{if A}_1\text{.type }\neq T_1 \text{ then} \\
& \quad \quad \quad \text{issue type error} \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{issue type error} \} \\
A \rightarrow E_1 & \{ \text{E}_1\text{.env = A.env;} \\
& \quad \text{A.type := E}_1\text{.type} \} \\
A \rightarrow E_1 , A_1 & \{ \text{E}_1\text{.env = A.env;} \\
& \quad \text{A.type := E}_1\text{.type} \times A_1\text{.type} \}
\end{align*}
\]
Implicit conversions (or “coercions”) occur as a result of applying semantic rules of the language, e.g., perhaps evaluating $r + i$, where $r$ is a real and $i$ is an integer, causes implicit conversion of the fetched value of $i$ to a real before the addition. This complicates type-checking:

$$E \rightarrow E_1 + E_2 \quad \{ \begin{array}{l}
E_1.env = E.env; \ E_2.env = E.env;
\end{array}$$

case $(E_1.type,E_2.type)$ of

$(\text{integer, integer})$: $E$.type := integer

$(\text{integer, real})$: 

$(\text{real, integer})$: 

$(\text{real, real})$: $E$.type := real

otherwise: issue type error

The relationship between integer and real is a special case of subtyping (more later).
A more compact way to specify typing rules is by using **inference rules** or **judgments** in the style of mathematical logic.

Each judgment for expressions has the form

\[ TE \vdash e : t \]

Intuitively this says that expression \( e \) has type \( t \), under the assumption that the type of each variable used in \( e \) is given by the **type environment** \( TE \).

We write \( TE(x) \) for the result of looking up \( x \) in \( TE \), and \( TE + \{ x \mapsto t \} \) for the type environment obtained from \( TE \) by extending it with a new binding from \( x \) to \( t \).

The key point is that an expression is well-typed **if-and-only-if** we can derive a typing judgment for it.
Here are some of the rules from the attribute-grammar formalism transformed into judgments.

\[
\frac{}{TE \vdash \text{num} : \text{integer}} \quad \text{(Num)}
\]

\[
\frac{id \in \text{dom}(TE)}{TE \vdash \text{id} : TE(id)} \quad \text{(Var)}
\]

\[
\frac{TE \vdash e_1 : t_1 \quad TE \vdash e_2 : t_2}{TE \vdash (e_1, e_2) : t_1 \times t_2} \quad \text{(Pair)}
\]

\[
\frac{TE \vdash e_1 : \text{integer} \quad TE \vdash e_2 : \text{integer}}{TE \vdash e_1 \ \text{div} \ e_2 : \text{integer}} \quad \text{(Div)}
\]

\[
\frac{TE \vdash e_1 : \text{boolean} \quad TE \vdash e_2 : \text{boolean}}{TE \vdash e_1 \ \text{or} \ e_2 : \text{boolean}} \quad \text{(Or)}
\]
MORE EXPRESSION RULES

\[ \frac{TE \vdash e_1 : \text{array}(t) \quad TE \vdash e_2 : \text{integer}}{TE \vdash e_1[e_2] : t} \]  
(Subscript)

\[ \frac{}{TE \vdash e : t_1 \times t_2} \quad \frac{}{TE \vdash e.fst : t_1} \]  
(Fst)

\[ \frac{TE \vdash e_1 : \text{integer} \quad TE \vdash e_2 : \text{integer}}{TE \vdash e_1 < e_2 : \text{boolean}} \]  
(LT)

\[ \frac{TE \vdash e_1 : \text{integer} \quad TE \vdash e_2 : \text{integer}}{TE \vdash e_1 = e_2 : \text{boolean}} \]  
(EQI)

\[ \frac{TE \vdash e_1 : \text{boolean} \quad TE \vdash e_2 : \text{boolean}}{TE \vdash e_1 = e_2 : \text{boolean}} \]  
(EQB)
Statement Rules

Judgments for statements omit the result environment, and simply assert that the statement is well-typed.

\[
\begin{align*}
\frac{TE \vdash e : t \quad TE(id) = t}{TE \vdash id := e} & \quad \text{(Assign)} \\
\frac{TE \vdash e : boolean \quad TE \vdash s}{TE \vdash \text{if } e \text{ then } s} & \quad \text{(If)} \\
\frac{TE \vdash s_1 \quad TE \vdash s_2}{TE \vdash s_1 ; s_2} & \quad \text{(Sequence)} \\
\vdash texp : t \quad TE \vdash \{id \mapsto t\} \vdash s \quad & \quad \text{(Decl)}
\end{align*}
\]
Judgments for type expressions just translate the external syntax for types into the internal representation:

\[
\frac{}{\vdash \text{bool} : \text{boolean}} \quad \text{(Bool)}
\]

\[
\frac{}{\vdash \text{int} : \text{integer}} \quad \text{(Int)}
\]

\[
\frac{\vdash \text{texp} : \text{t}}{\vdash \text{array of } \text{texp} : \text{array(}t\text{)}} \quad \text{(Array)}
\]

\[
\frac{\vdash \text{texp}_1 : \text{t}_1 \quad \vdash \text{texp}_2 : \text{t}_2}{\vdash \text{pair } \text{texp}_1 \ \text{texp}_2 : \text{t}_1 \times \text{t}_2} \quad \text{(Pair)}
\]
When do two identifiers have the "same" type, or "compatible" types?

E.g., if \(a\) has type \(t_1\), \(b\) has type \(t_2\) and \(f\) has type \(t_2 \rightarrow t_3\), how must \(t_1\) and \(t_2\) be related for these to make sense?

\[
a := b \quad f(a)
\]

To maintain **type safety** we must insist at a minimum that \(t_1\) and \(t_2\) are **structurally equivalent**.

Structural equivalence is defined inductively:

- Primitive types are equivalent iff they are exactly the same type.
- Cartesian product types are equivalent if their corresponding component types are equivalent. (Record field names are typically ignored.)
- Disjoint union types are equivalent if their corresponding component types are equivalent.
- Mapping types (arrays and functions) are the same if their domain and range types are the same.
Another way to say this: two types are equal if they have the same set of values.

Recursive types are a challenge. Are these two types structurally equivalent?

\[
\text{type } t1 = \{ \text{a: int, b: POINTER TO t1} \}; \\
\text{type } t2 = \{ \text{a: int, b: POINTER TO t2} \};
\]

Intuitively yes, but it’s (a little) tricky for a type-checking algorithm to determine this!
Question of equivalence is more interesting if language has type **names**, which arise for two main reasons:

- As a convenient shorthand to avoid giving the full type each time. E.g.,

  ```plaintext
  function f(x:int * bool * real) : int * bool * real = ...
  type t = int * bool * real
  function f(x:t) : t = ...
  ```

- As a way of improving program correctness by subdividing values into types according to their meaning **within the program**.

  ```plaintext
  type polar = { r:real, a:real }
  type rect = { x:real, y:real }
  function polar_add(x:polar,y:polar) : polar ...
  function rect_add(x:rect,y:rect) : rect ...
  var a:polar; c:rect;
  a := (150.0,30.0) (* ok *)
  polar_add(a,a) (* ok *)
  c := a (* type error *)
  rect_add(a,c) (* type error *)
  ```

For this to be useful, some structurally equivalent types must be treated as **inequivalent**.
NAME EQUIVALENCE

Simplistic idea: Two types are equivalent iff they have the same name.

Supports polar/rect distinction.

But pure name equivalence is very restrictive, e.g.:

```plaintext
type ftemp = real
type ctemp = real
var x:ftemp, y:ftemp, z: ctemp;
x := y; (* ok *)
x := 10.0; (* probably ok *)
x := z; (* type error *)
x := 1.8 * z + 32.0; (* probably type error *)
```

Different types now seem too distinct; can’t even convert from one form of real to another.
Also: what about unnamed type expressions?

```plaintext
type t = int * int
procedure f(x: int * int) = ...
procedure g(x: t) = ...
var a:t = (3,4)
g(a); (* ok *)
f(a); (* ok or not ?? *)
```

Because of these problems with pure name equivalence, most languages use **mixed** solutions.
C uses structural equivalence for array and function types, but name equivalence for `struct`, `union`, and `enum` types. For example:

```c
char a[100];
void f(char b[]);
f(a); /* ok */

struct polar{float x; float y;};
struct rect{float x; float y;};
struct polar a;
struct rect b;
a = b; /* type error */
```

A type defined by a `typedef` declaration is actually just an abbreviation for an existing type. Note that this policy makes it easy to check equivalence of recursive types, which can only be built using structs.

```c
struct fred {int x; struct fred *y;} a;
struct bill {int x; struct fred *y;} b;
a = b; /* type error */
```
ML uses structural equivalence, except that each `datatype` declaration creates a new type unlike all others.

```ml
datatype polar = POLAR of real * real
datatype rect = RECT of real * real
val a = POLAR(1.0,2.0) and b = RECT(1.0,2.0)
if (a = b) ... (* type error *)
```

Note that the mandatory use of constructors makes it possible to uniquely identify the types of literals.

Note that a datatype need not declare a record:

```ml
datatype fahrenheit = F of real
datatype celsius = C of real
val a = F 150.0 and b = C 150.0
if (a = b) ... (* type error *)
fun convert(F x) = C(1.8 * x + 32.0) (* ok *)
```

For type abbreviation, ML offers the `type` declaration, which simply gives a new name for an existing type.

```ml
type centigrade = celsius
fun g(x:centigrade) = if x = b ... (* ok *)
```
Java uses nearly strict name equivalence, where names are either:

- One of eight built-in **primitive** types (int, float, boolean, etc.), or
- Declared classes or interfaces (**reference** types).

The only non-trivial type expressions that can appear in a source program are **array** types, which are compared structurally, using name equivalence for the ultimate element type. Java has no mechanism for type abbreviations.

Java types form a **subtyping** hierarchy:

- If class $A$ extends class $B$, then $A$ is a subtype of $B$.
- If class $A$ implements interface $I$, then $A$ is a subtype of $I$.
- If numeric type $t$ can be coerced to numeric type $u$ without loss of precision, then $t$ is a subtype of $u$.

If $T_1$ is a subtype of $T_2$, then a value of type $T_1$ can be used wherever a value of $T_2$ is expected.