Consider a simple language of declarations, statements, and expressions.

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

Actions for declarations synthesize environment attributes:

- \[ D \rightarrow \epsilon \{ D.env := empty \} \]
- \[ D \rightarrow id : T_1 ; D_1 \{ D.env := extend(D_1.env,binding(id, T_1.type)) \} \]
- \[ T \rightarrow bool \{ T.type := boolean \} \]
- \[ T \rightarrow int \{ T.type := integer \} \]
- \[ T \rightarrow array of T_1 \{ T.type := array(T_1.type) \} \]
- \[ T \rightarrow pair T_1 T_2 \{ T.type := T_1.type \times T_2.type \} \]
Expressions

Actions for expressions **check** for compatible operands and **synthesize** attribute type:

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

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

\[
S \rightarrow \text{id} := E_1 \quad \{ \ E_1.env = S.env; \newline
\text{if } E_1.type \neq \text{lookup}(S.env, \text{id}) \text{ then } \newline\text{issue type error } \} \\
(Must also check that \text{id} is an l-value \text{ that can be assigned into.})
\]

\[
S \rightarrow \text{if } E_1 \text{ then } S_1 \quad \{ \ E_1.env = S.env; S_1.env = S.env; \newline\text{if } E_1.type \neq \text{boolean} \text{ then } \newline\text{issue type error } \}
\]

\[
S \rightarrow S_1 ; S_2 \quad \{ \ S_1.env = S.env; S_2.env = S.env; \}
\]
PROCEDURE/FUNCTION DEFINITIONS AND CALLS

Can describe type of function as $type_1 \times type_2 \times \ldots \times type_n \rightarrow type$

\[
D \rightarrow \text{id} ( F_1 ) : T_1 ; D_1 \quad \{ \text{D.env := extend(D_1.env, binding(id,F_1.type \rightarrow T_1.type)) } \}
\]

\[
F \rightarrow \text{id} : T_1 \quad \{ \text{F.type := T_1.type} \}
\]

\[
F \rightarrow \text{id} : T_1 , F_1 \quad \{ \text{F.type := T_1.type } \times F_1.type \}
\]

\[
E \rightarrow \text{id} ( A_1 ) \quad \{ \text{A_1.env = E.env;}
\text{if lookup(E.env,id) = T_1 \rightarrow T_2 then}
\text{if A_1.type } \neq T_1 \text{ then}
\text{issue type error}
\text{E.type := T_2}
\text{else}
\text{issue type error } \}
\]

\[
A \rightarrow E_1 \quad \{ \text{E_1.env = A.env;}
\text{A.type := E_1.type } \}
\]

\[
A \rightarrow E_1 , A_1 \quad \{ \text{E_1.env = A.env;}
\text{A.type := E_1.type } \times A_1.type \}
\]
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:

$$E \rightarrow E_1 + E_2 \quad \{ \quad E_1.env = E.env; \ E_2.env = E.env; \quad$$

$$\quad \text{case (E}_1\text{.type,E}_2\text{.type) of}$$

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

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

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

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

$$\quad \text{otherwise: issue type error } \}$$

The relationship between integer and real is a special case of subtyping (more later).
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 \( t_2 \to t_3 \), how must \( t_1 \) and \( t_2 \) be related for these to make sense?

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

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?

```haskell
type t1 = { a:int, b: POINTER TO t1 };
type t2 = { 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.,

  ```
  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.

  ```
  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.:

```
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 \texttt{datatype} declaration creates a new type unlike all others.

\begin{verbatim}
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 *)
\end{verbatim}

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

Note that a \texttt{datatype} need not declare a record:

\begin{verbatim}
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 *)
\end{verbatim}

For type abbreviation, ML offers the \texttt{type} declaration, which simply gives a new name for an existing type.

\begin{verbatim}
type centigrade = celsius
fun g(x:centigrade) = if x = b ... (* ok *)
\end{verbatim}
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.