Type Systems.

- A "type system" is a collection of rules for assigning type expressions to various parts of a program.
- A type checker implements a type system.

Static and Dynamic Checking of Types.

- Checking done by a compiler is said to be static, while checking done when the target program runs is termed dynamic.
- In principle, any check can be done dynamically, if the target code carries the type of an element along with the value of that element.
- A "sound" type system eliminates the need for dynamic checking for type systems because it allows to determine statically that these errors cannot occur when the target program runs.
- A language is "strongly typed" if its compiler can guarantee that the programs it accept execute without type errors.

Error Recovery

- At the very least, the compiler must report the nature and location of the error.
- It is desirable for the type checker to recover from errors, so it can check the rest of the input.

Specification of a Simple Type Checker.

- Here, we specify a type checker for a simple language in which the type of each identifier must be declared before the identifier is used.
- The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions.

A Simple Language

- The grammar in Fig.1 generates programs, represented by the nonterminal P, consisting of a sequence of declarations D followed by a single expression E.

```
P → D ; E
D → D ; D | id : T
T → int | char | array [num] of T | ↑T
E → literal | num | id | E mod E | E [E] | E↑
```

Fig.1.

- One program generated by the above grammar is

```
key : int ;
key mod 1999
```

- In the translation scheme of Fig.2, the action associated with the production, D → id : T, saves a type in a symbol table entry for an identifier. The action "addtype (id.entry, T.type)" is applied to synthesized attribute "entry" pointing to the symbol table entry for id and a type expression represented by synthesized attribute "type" of nonterminal T.

```
P → D ; E
D → D ; D
D → id : T      { addtype (id.entry, T.type); }
T → char        { T.type = char; }
T → int         { T.type = integer; }
T → array [num] of T1   { T.type = array (1..num.val, T1.type); }
T → ↑T          { T.type = pointer (T1.type); }
```

Fig.2. Translation scheme that saves the type of an identifier

Type Checking of Expressions

- In the following rules, the synthesized attribute "type" for E gives the type expression assigned by the type system to the expression generated by E.

E → literal { E.type = char; }

E → num { E.type = integer; }

E → id { E.type = lookup (id.entry); }

E → E_1 mod E_2 { E.type = if E_1.type = integer and E_2.type = integer then integer else type-error; }

E → E_1 [E_2] { E.type = if E_1.type = array (s, t) and E_2.type = integer then t else type-error; }

E → E↑ { E.type = if E_1.type = pointer (t) then t else type-error; }

- To allow identifiers to have type "boolean", we can introduce the production

  T → boolean

  to the grammar of Fig. 1.

- The introduction of comparison operators like "<" and logical connectives like "and" into the productions for E would allow the construction of expressions of type boolean.