

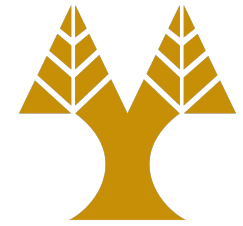
# ΕΠΛ323 - Θεωρία και Πρακτική Μεταγλωττιστών

Lecture 10a

## **Type Checking**

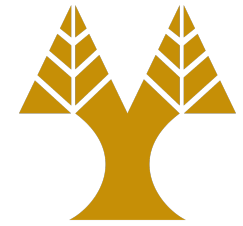
Elias Athanasopoulos  
eliasathan@cs.ucy.ac.cy

# Static Checking



- Ensures that certain kinds of programming errors will be detected and reported **at compile-time**:
  - *Type checks*. An array variable and a function variable are added together.
  - *Flow-of-control checks*. A `break` statement in C causes control to leave the smallest enclosing `while`, `for`, or `switch` statement, while the smallest enclosing statement does not exist.
  - *Uniqueness checks*. Labels in a `case` statement must be distinct.
  - *Name-related checks*. In Ada, a loop or block may have a name that appears in the beginning and end of the construct.

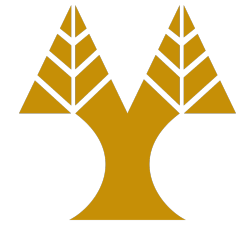
# Dynamic Checking



- Checks performed by the program **at run-time**.
- In principle, all checks can be performed at run-time, but this is not efficient.
  - A sound type system allows us to determine statically that these errors cannot occur when the target program runs.
- A language is strongly typed if its compiler can guarantee that a compiled program will execute without errors.
- Not always possible.

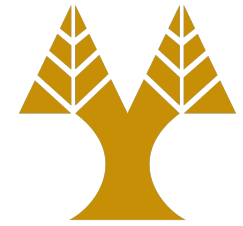
```
table: array[0..255] of char;  
i: integer  
...  
x := table[i];
```

# Type Systems



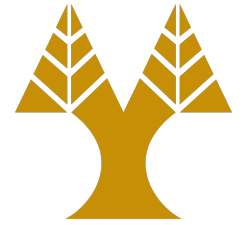
- Based on information about the syntactic constructs, the notion of types, and the rules for assigning types to language constructs.
  - “If both operands of the arithmetic operators of addition, subtraction and multiplication are of type integer, then the result is of type integer.”
  - “The result of the unary & operator is a pointer of the object referred to by the operand. If the type of the operand is `...`, the type of the result is `pointer to...`.”

# Basic and Constructed Types



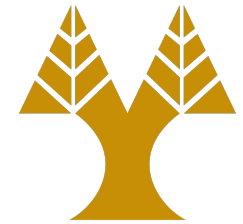
- Basic types are atomic types with no internal structure,
  - C: char, int, float, double, etc.
  - Pascal: boolean, character, integer, real, ranges (1..10), enums, etc.
- Constructed types,
  - C: struct, arrays.
  - Pascal: arrays, records, sets.

# Type Expressions



- Basic type (boolean, char, integer, and real), and special basic types, *type\_error*, which signals an error during type checking, and *void*, which denotes the absence of value.
- Type names.
- Type constructors.
  - *Arrays*.
  - *Products*.
  - *Records*.
  - *Pointers*.
  - *Functions*.
- Variables holding values that are type expressions.

# Simple Type Checker



```

$$\begin{aligned} P &\rightarrow D ; E \\ D &\rightarrow D ; D \mid \text{id} : T \\ T &\rightarrow \text{char} \mid \text{integer} \mid \text{array} [ \text{num} ] \text{ of } T \mid \wedge T \\ E &\rightarrow \text{literal} \mid \text{num} \mid \text{id} \mid E \text{ mod } E \mid E [ E ] \mid E \wedge \end{aligned}$$

```

## Examples

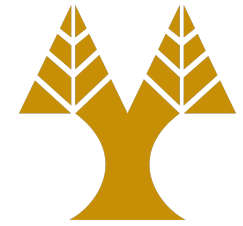
```
key: integer;
```

```
key mod 1999
```

```
array [256] of char
```

```
^integer
```

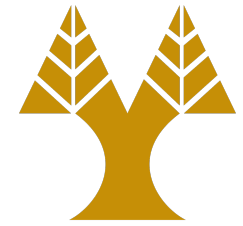
# Translation Scheme



```
E → D ; E
D → D ; D
D → id : T           { addtype(id.entry, T.type) }
T → char             { T.type := char }
T → integer         { T.type := integer }
T →  $\hat{T}_1$            { T.type := pointer(T}_1.type) }
T → array [num] of T}_1
      { T.type := array(1..num.val, T}_1.type) }
```

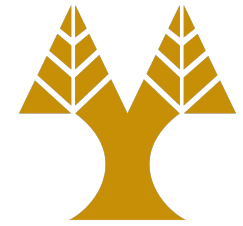


# Type Checking of Expressions



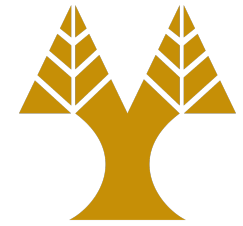
```
E → literal    { E.type := char }
E → num        { E.type := integer }
E → id         { E.type := lookup(id.entry) }
E → E1 mod E2 { E.type :=
    if E1.type = integer and
    E2.type = integer then integer
    else type_error }
E → E1[E2]   { E.type :=
    if E2.type = integer and
    E1.type = array(s,t) then t
    else type_error }
E → E1^      { E.type :=
    if E1.type = pointer(t) then t
    else type_error }
```

# Type Checking of Statements



```
S → id : E           { S.type :=  
                        if id.type = E.type then void  
                        else type_error }  
S → if E then S1 { S.type :=  
                  if E.type = boolean then S1.type  
                  else type_error }  
S → while E do S1 { S.type :=  
                    if E.type = boolean then S1.type  
                    else type_error }  
T → S1 ; S2       { S.type :=  
                    if S1.type = void and  
                      S2.type = void then void  
                    else type_error }
```

# Type Checking of Functions



```
E → E (E)      { S.type :=  
                  if id.type = E.type then void  
                  else type_error }  
  
T → T1 '→' T2  { T.type := T1.type → T2.type }  
  
E → E1(E2)    { E.type :=  
                  if E2.type = s and  
                    E1.type = s→t then t  
                  else type_error }
```