Introduction

- A source program should follow both the syntactic and semantic rules of the source language.

- Some rules can be checked ***statically*** during compile time and other rules can only be checked ***dynamically*** during run time.
- Static checking includes the syntax checks performed by the parser and semantic checks such as type checks, flow-of-control checks, uniqueness checks, and name-related checks.

- Here we focus on type checking.

Use of Type

- Virtually all high-level programming languages associate types with values.
- Types often provide an implicit context for operations.
  - In C the expression x + y will use integer addition if x and y are int's, and floating-point addition if x and y are float's.
- Types can catch programming errors at compile time by making sure operators are applied to semantically valid operands.
  - For example, a Java compiler will report an error if x and y are String's in the expression x * y.

Types

- ***Basic types*** are atomic types that have no internal structure as far as the programmer is concerned.
  - They include types like **`integer, real, boolean`**, and **`character`**.
  - Subrange types like **`1..10`** in Pascal and enumerated types like **`(violet, indigo, blue, green, yellow, orange, red)`** are also basic types.
- ***Constructed types*** include **arrays**, **records**, **sets**, and **structures** constructed from the basic types and/or other constructed types.
  - **Pointers** and **functions** are also constructed types.

Type Expressions

- **Type Expressions** denote the type of a language construct
  - It is either a basic type or formed from other type expressions by applying an operator called a ***type constructor***.
    - Example: a function from an integer to an integer
    - A type constructor applied to a type expression is a type expression.

- Here we use type expressions formed from the following rules:
  - A basic type is a type expression. Other basic type expressions are **type-error** to signal the presence of a type error and **void** to signal the absence of a value.

  - If a type expression has a name then the name is also a type expression.

Type Constructors

- *Arrays*. If T is a type expression and I is the type expression of an index set then *array* (I, T) denotes an array of elements of type T.
- *Products*. If T_1 and T_2 are type expressions, then their Cartesian product, T_1 x T_2, is a type expression.
  - For example if the arguments of a function are two reals followed by an integer then the type expression for the arguments is: **real** x **real** x **integer**.
- *Records*. The fields in a record (or structure) have names which should be included in the type expression of the record. The type expression of a record with n fields is:

$$record\ (F_1 \text{ x } F_2 \text{ x } \ldots \text{ x } F_n\ )$$

where if the name of field i is name_i and the type expression of field i is T_i then F_i is:

$$(\text{name}_i \text{ x } T_i\ ).$$

Type Constructors

- *Pointers*. If *T* is a type expression then *pointer* (*T*) denotes a pointer to an object of type *T*.
- *Functions*. A function maps elements from its *domain* to its *range*. The type expression for a function is: *D* --> *R* where *D* is the type expression for the domain of the function and *R* is the type expression for the range of the function. For example, the type expression of the **mod** operator in Pascal is: **integer** x **integer** --> **integer** because it divides an integer by an integer and returns the integer remainder.
  - The type expression for the domain of a function with no arguments is **void** and the type expression for the range of a function with no returned value is **void**: e.g., **void** --> **void** is the type expression for a procedure with no arguments and no returned value.

Type Systems

- A type system is a set of rules for assigning type expressions to the syntactic constructs of a program and for specifying
  - ***type equivalence*** - when the types of two values are the same,
  - ***type compatibility*** - when a value of a given type can be used in a given context
  - ***type inference*** - rules that determine the type of a language construct based on how it is used.

Type Equivalence

- Forms of type equivalence Name equivalence: two types are equivalent iff they have the same name.
- Structural equivalence: two types are equivalent iff they have the same structure.
- To test for structural equivalence, a compiler must encode the structure of a type in its representation. A tree (or type graph) is typically used.

Type Checker

- Most all programming languages insist that the type of an ID token be declared before it can be used.
- A type checker makes sure that a program obeys the type-compatibility rules of the language.
- We can think about types in several different ways:
  - Denotational: a type is a set of values called a domain.
  - Constructive: a type is either a primitive type or a composite type created by applying a type constructor to simpler types.
  - Abstraction-based: a type is an interface consisting of a set of operations with well-defined and mutually consistent semantics.

Typing in Programming Languages

- The type system of a language determines whether type checking can be performed at compile time (*statically*) or at run time (*dynamically*).
- A statically typed language is one in which all constructs of a language can be typed at compile type.
  - C, ML, and Haskell are statically typed.
- A dynamically typed language is one in which some of the constructs of a language can only be typed at run time.
  - Perl, Python, and Lisp are dynamically typed.
- A strongly typed language is one in which the compiler can guarantee that the programs it accepts will run without type errors.
  - ML and Haskell are strongly typed.
- A type-safe language is one in which the only operations that can be performed on data in the language are those sanctioned by the type of the data.

Type Inference Rules

- Type inference rules specify for each operator the mapping between the types of the operands and the type of the result.
- E.g., result types for x + y:

+	**int**	**float**
int	int	float
float	float	float

- Operator and function overloading
  - In Java the operator + can mean addition or string concatenation depending on the types of its operands.
  - We can choose between two versions of an overloaded function by looking at the types of their arguments

Type Inference Rules - Functions

- Compiler must check that the type of each actual parameter is compatible with the type of the corresponding formal parameter.
- It must check that the type of the returned value is compatible with the type of the function.
- The *type signature* of a function specifies the types of the formal parameters and the type of the return value.
- Example: strlen in C
  - Function prototype in C:

    unsigned int strlen(const char *s);
  - Type expression:

    strlen: const char * → unsigned int

Type Inference Rules - Polymorphism

- A polymorphic function allows a function to manipulate data structures regardless of the types of the elements in the data structure
- Example: an ML program for the length of a list

```
fun length(x) =
        if null(x) then 0 else length(tl(x))+1;
```

Type Conversions

- Implicit type conversions
  - In an expression such as f + i where f is a float and i is an integer, a compiler must first convert the integer to a float before the floating point addition operation is performed. That is, the expression must be transformed into an intermediate representation like

    t1 = INTTOFLOAT i

    t2 = x FADD t1
- Explicit type conversions
  - In C, explicit type conversions can be forced ("coerced") in an expression using a unary operator called a cast. E.g., sqrt((double) n) converts the value of the integer n to a double before passing it on to the square root routine sqrt.