Types

• Organization of **untyped values**
  – Untyped universes: bit strings, S-expr, ...
  – Categorize based on usage and behavior

• Type = set of computational entities with uniform behavior

• Constraints to enforce correctness
  – Check the applicability of operations
    • Should not try to multiply two strings
    • Should not use a character value as a condition of an if-statement
    • Should not use an integer as a pointer
Examples of Type Checking

• Built-in operators should get operands of correct types
• Type of left-hand side must agree with the value on the right-hand side
• Procedure calls: check number and type of actual arguments
• Return type should match returned value
Static Typing

• Statically typed languages: expressions in the code have **static types**
  – static type = claim about run-time values
  – Types are either **declared** or **inferred**
  – Examples: C, C++, Java, ML, Pascal, Modula-3

• A statically typed language typically does some form of **static type checking**
  – E.g., at compile time Java checks that the `[]` operator is applied to a value of type “array”
  – May also do dynamic (run-time) checking
    • e.g., Java checks at run time for array indices out of bounds and for null pointers
Dynamic Typing

• Dynamically-typed languages: entities in the code do not have static types
  – Examples: Lisp, Scheme, CLOS, Smalltalk, Perl, Python
  – Entities in the code do not have declared types, and the compiler does not try to infer types for them

• Dynamic type checking
  – Before an operation is performed at run time
  – E.g., in Scheme: \((+ 5 \#t)\) fails at run time, when the evaluation expects to see two numeric atoms as operands of +
Strongly vs. Weakly Typed

• **Strongly typed** languages: type-incorrect operations are not performed at run time
  – Things cannot “go wrong”: no undetected type errors
  – Certain run-time errors are possible but clearly marked as such
    • i.e. array index out of bounds, null pointer
  – C/C++: weakly typed, Java: strongly typed

• **Independent of static vs. dynamic**
  – Lisp, Scheme, Python: strongly, dynamically typed
  – Forth: weakly, dynamically typed
Examples of Types

• Integers
• Arrays of integers
• Pointers to integers
• Records with fields \texttt{int x} and \texttt{int y}
  – e.g., “struct” is C
• Objects of class C or a subclass of C
  – e.g., C++, Java, C#
• Functions from any list to integers
Numeric Types

• Varied from language to language
• C does not specify the ranges of numeric types
  – Integer types: char, short, int, long, long long
    • Includes “unsigned” versions of these
  – Floating-point types: float, double, long double
• Java specifies the ranges of numeric types
  – byte: 8-bit signed two's complement integer [-128,+127]
  – short: 16-bit signed two's complement integer [-32768,+32,767]
  – int: 32-bit signed two's complement integer [-2147483648,+2147483647]
  – long: 64-bit signed two's complement integer
    [-9223372036854775808, +9223372036854775807]
  – float/double: single/double-precision 32-bit IEEE 754 floating point
  – char: single 16-bit Unicode character; minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65535)
Enumeration Types

• C: a set of named integer constant values
  – Example from the C specification
    ```c
    enum hue { chartreuse, burgundy, claret=20, winedark };  
    /* the set of integer constant values is { 0, 1, 20, 21 } */ 
    ```
    ```c
    enum hue col, *cp;
    col = claret; cp = &col;
    if (*cp != burgundy) ...
    ```

• Java: a fixed set of named items (not integers)
  ```java
  enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }
  ```
  – In reality, it is like a class: e.g., it can contain methods
Types as Sets of Values

• Integers
  – Any number than can be represented in 32 bits in signed two’s-complement
  – “type int” = { -2^{31}, ..., 2^{31} - 1 }

• Class type (not the same as a class)
  – Any object of class C or a subclass of C
  – “type C” = set of all instances of C or of any transitive subclass of C ("class C" is just a blueprint for objects)

• Subtypes are subsets: T2 is a subtype of T1 if the T2’s set of values is a subset of T1’s set of values
Monomorphism vs. Polymorphism

• Greek:
  – mono = single
  – poly = many
  – morph = form

• Monomorphism
  – Every computational entity belongs to exactly one type

• Polymorphism
  – A computational entity can belong to multiple types
Types of Polymorphism

- parametric
- inclusion (subset)
- overloading
- coercion

polymorphism

universal

ad hoc
Coercion

• Values of one type are silently converted to another type
  – e.g. addition: $3.0 + 4$ : converts 4 to 4.0
    • $\text{int} \times \text{int} \rightarrow \text{int}$ or $\text{real} \times \text{real} \rightarrow \text{real}$

• In a context where the type of an expression is not appropriate
  – either an automatic coercion (conversion) to another type is performed automatically
  – or if not possible: compile-time error
Coercions

• Widening
  – coercing a value into a “larger” type
  – e.g., \texttt{int} to \texttt{float}, subclass to superclass

• Narrowing
  – coercing a value into a “smaller” type
  – loses information, e.g., \texttt{float} to \texttt{int}
Widening Primitive Conversions in Java

• Widening primitive conversions
  – byte to short, int, long, float, or double
  – short to int, long, float, or double
  – char to int, long, float, or double
  – int to long, float, or double
  – long to float or double
  – float to double

• “integral type to integral type” and “float to double” do not lose any information
Widening Primitive Conversions in Java

• Language Spec says
  – Conversion of an int or long value to float, or of a long value to double, may result in loss of precision
  – The result may lose some of the least significant bits of the value. In this case, the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode
Contexts for Widening Conversions

• **Assignment conversion**: when the value of an expression is assigned to a variable

• **Method invocation conversion**: applied to each argument value in a method or constructor invocation
  – The type of the argument expression must be converted to the type of the corresponding formal parameter

• **Casting conversion**: applied to the operand of a cast operator: (float) 5
Contexts for Widening Conversions

• **Numeric promotion**: converts operands of a numeric operator to a common type

• Example: binary numeric promotion
  – e.g. +, -, *, etc.
  – If either operand is double, the other is converted to double
  – Otherwise, if either operand is of type float, the other is converted to float
  – Otherwise, if either operand is of type long, the other is converted to long
  – Otherwise, both are converted to type int
Narrowing Conversions

• Narrowing primitive conversions in Java
  – e.g. long to byte, short, char, or int
  – float to byte, short, char, int, or long
  – double to byte, short, char, int, long, or float

• Examples of loss of information
  – int to short loses high bits
  – int not fitting in byte changes sign and magnitude
  – double too small for float underflows to zero
Polymorphism by Overloading

• Multiple definitions of the same name
• E.g. name + for several operations: has several types (function types $X \to Y$)
  – double $\times$ double $\to$ double (binary plus)
  – float $\times$ float $\to$ float
  – long $\times$ long $\to$ long
  – int $\times$ int $\to$ int
  – double $\to$ double (unary plus)
  – float $\to$ float
  – long $\to$ long
  – int $\to$ int
Overloading vs. Overriding in Java

class **Point** {
    int x = 0, y = 0;
    void **move**(int dx, int dy) { x += dx; y += dy; } }

class **RealPoint** extends **Point** {
    float x = 0.0, y = 0.0;
    void **move**(int dx, int dy)
        { move((float)dx, (float)dy); } 
    void **move**(float dx, float dy)
        { x += dx; y += dy; } }
Overloading vs. Overriding in Java

```java
public static void main(String[] args) {
    RealPoint rp = new RealPoint();
    // compile-time resolution: the most specific
    // target method
    rp.move(1.5f, 1.5f);  → RealPoint.move(float,float)
    rp.move(2,2);  → RealPoint.move(int,int)
    Point p = rp;
    p.move(3,3);  → compile time: Point.move(int,int)
    → run time: RealPoint.move(int,int)
    // can we say p.move(3.3f,3.3f)?
}
```
Overloading: Most Specific Method

class Test {
    static void test(RealPoint p, Point q) { ... }
    static void test(Point p, RealPoint q) { ... }
    public static void main(String[] args) {
        RealPoint rp = new RealPoint();
        test(rp, rp); // compile-time error
    }
}
Parametric Polymorphism: Generics in Ada

generic
    type T is private;

function Id(X : in T) return T is
begin
    return X;
end;

function IntId is new Id (INTEGER);
function FloatId is new Id (FLOAT);

Similar: templates in C++; generics in Java 1.5 and later

generic functions and classes
package java.util;

public interface Set<E> extends Collection<E> { …
    Iterator<E> iterator();
    boolean add(E e);
    boolean addAll(Collection<? extends E> c); }

class Rectangle { … }

class SwissRectangle extends Rectangle { … }

Set<Rectangle> s = new HashSet<Rectangle>();
    s.add(new Rectangle(1.,2.)); s.add(new SwissRectangle(3.,4.,5));
Set<SwissRectangle> s2 = new TreeSet<SwissRectangle>();
    s2.add(new SwissRectangle(6.,7.,8)); s.addAll(s2);
Inclusion (Subset) Polymorphism

• Subtype relationships among types
  – Defined by “Y is subset of X” (i.e., set inclusion)
• A computational entity of a subtype may be used in any context that expects an entity of a supertype
• Typical examples
  – Imperative languages: record types
  – Object-oriented languages: class types
Subtyping in Java

• Recall that **class type C** is the set of all instances of class C or of any transitive subclass of C

• Subtyping between class types
  
  ```java
  class X { int m () { ... } }
  class Y extends X { int m () { ... } }
  X x = new Y();
  int i = x.m();
  ```

• Interface type: the set of all instances of classes that implements the interface (transitively)
  
  ```java
  interface Z { bool m(); }
  class W implements Z { bool m() { ... } }
  Z z = new W(); bool b = z.m();
  ```
Example: The Core Interpreter

• To illustrate these issues, consider again the implementation of the interpreter

• Use of PT array or PT data type
  – The compiler will not stop us from creating a <decl> with a child <stmt>, instead of <id-list>
  – Conceptually, this is a type error: but our program does not declare “rich enough” types to catch such an error

• Solution: create a separate type for each non-terminal (e.g., a separate class type)
class Prog {
    private: DeclSeq* decl_seq; StmtSeq* stmt_seq;
    public:
        Prog() { decl_seq = NULL; stmt_seq = NULL; }
        void parse() {
            scanner->match(PROGRAM);
            decl_seq = new DeclSeq(); decl_seq->parse();
            scanner->match(BEGIN);
            stmt_seq = new StmtSeq(); stmt_seq->parse();
            scanner->match(END); scanner->match(EOF);
        }
        void print() {
            cout << "program "; decl_seq->print();
            cout << " begin "; stmt_seq->print(); cout << " end";
        }
        void exec() {
            decl_seq->exec(); stmt_seq->exec();
        }
};
class StmtSeq {
    private: Stmt* stmt; StmtSeq* stmt_seq;
    public:
        StmtSeq() { stmt = NULL; stmt_seq = NULL; }
        void parse() {
            stmt = new Stmt(); stmt->parse();
            if (scanner->currentToken() == END) return;
            // Same for ELSE, ENDIF, ENDWHILE
            stmt_seq = new StmtSeq(); stmt_seq->parse();
        }
        void print() {
            stmt->print();
            if (stmt_seq != NULL) stmt_seq->print();
        }
        void exec() {
            stmt->exec();
            if (stmt_seq != NULL) stmt_seq->exec();
        }
};
Another Example

• $\langle \text{exp} \rangle ::= \text{int} \mid \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle \ast \langle \text{exp} \rangle$
  – Not Core, but similar to parts of Core

• For each expression, want to be able to
  – Compute its value: \texttt{int evalExpr()}
  – Determine if its value is even: \texttt{bool isEven()}