Types in Programming Languages

- Chapter 5, Sections 5.5 and 5.6
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 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
  - 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: expressions in the code do not have static types
  - Examples: Lisp, Scheme, CLOS, Smalltalk, Perl, Python

- Dynamic type checking
  - Before an operation is performed at run time
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: strongly, dynamically typed
  - Forth: weakly, dynamically typed
Examples of Types

- Integers
- Arrays of integers
- Pointers to integers
- Records with fields `int x` and `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
Types as Sets of Values

- **Integers**
  - Any number than can be represented in 32 bits in signed two’s-complement
  - \( \text{int} = \{ -2^{31}, \ldots, 2^{31} - 1 \} \)

- **Class type**
  - Any object of class \( C \) or a subclass of \( C \)
  - \( C = \) set of all instances of \( C \) or of any transitive subclass of \( C \)

- **Subtypes are subsets**
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
Polymorphism by Overloading

- Multiple definitions of the same name
- E.g. name + for several operations: has several types (function types $X \rightarrow Y$)
  - double $\times$ double $\rightarrow$ double (binary plus)
  - float $\times$ float $\rightarrow$ float
  - long $\times$ long $\rightarrow$ long
  - int $\times$ int $\rightarrow$ int
  - double $\rightarrow$ double (unary plus)
  - float $\rightarrow$ float
  - long $\rightarrow$ long
  - int $\rightarrow$ int
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
generic functions and classes
Inclusion Polymorphism

- Subtype relationships among types
- 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

- Subtyping between class types

```java
class B { int foo () { ... } }
class C extends B { int foo () { ... } }
B p = new C();
int i = p.foo();
```

- Interface types
  - interface X { bool bar(); }
  - class A implements X { bool bar() { ... } }
  - X x = new A(); bool b = x.bar();
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: DS* ds; SS* ss;
public:
  Prog() { ds = NULL; ss = NULL; }
  void parseProg() {
    // check and read token PROGRAM
    ds = new DS(); ds->parseDS();
    // check and read token BEGIN
    ss = new SS(); ss->parseSS();
    // check and read END and ENDOFFILE
  }
  void printProg() {
    cout << "program"; ds->printDS();
    cout << "begin"; ss->printSS(); cout << "end;";
  }
  void execProg() {
    ds->execDS(); ss->execSS();
  }
};
class SS {
private: Stmt* s; SS* ss;
public:
    SS() { s = NULL; ss = NULL; }
    void parseSS() {
        s = new Stmt(); s->parseStmt();
        // if the current token is END, return. // otherwise, do error checking.
        ss = new SS(); ss->parseSS();
    }
    void printSS() {
        s->printStmt();
        if (ss == NULL) return; ss->printSS();
    }
    void execSS() {
        s->execStmt();
        if (ss == NULL) return; ss->execSS();
    }
};
Another Example

- \(<\text{exp}> ::= \text{int} \mid <\text{exp}> + <\text{exp}> \mid <\text{exp}> \ast <\text{exp}>\)
  - Not Core, but similar to parts of Core

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

- What classes should we have? What are the methods in these classes? What are the bodies of those methods?

- \text{id ::= <exp> - how about this one?}