Modular C Programming Example …

```
/* cube.h */
extern double cube(double x);
```

```
/* cube.c */
#include <stdio.h>
double cube(double x) {
  double r = x * x * x;
  printf("cube: %f --> %f\n", x, r);
  return r; }
```

```
/* cubing.c */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[]) {
  int k = atoi(argv[1]);
  printf("cubing: %d --> %d\n", k, cube(k));
  return 0;
}
```

Undetected Run-Time Type Error in C

```
#include <stdio.h>

union utag {int a;
             float p;
            } u;

void main() {
  float x = 2.0;
  u.a = 2135329191;
  printf("%d   %f\n", u.a, x + u.p);
}
```

```
2135329191   263901874997436424049023275123576143872.000000
```

- interpretation as **float** values is **not well-defined** for **int** values
- interpretation as **float** values is ***possible*** for **int** values
- **unexpected values can produce undetected misbehaviour**

Detected Run-Time Type Error in Java

```
class Point { protected double _x, _y;
  public Point(double x, double y) { _x = x; _y = y; }
  public String toString() { return "(" + _x + ", " + _y + ")"; }
}
class Point3 extends Point { protected double _z;
  public Point3(double x, double y, double z) { super(x,y); _z = z; }
  public void up(double dz) {_z += dz; }
  public String toString() { return "(" + _x + ", " + _y + ", " + _z + ")"; }
}
class DownCastError {
  public static void main(String[] args) {
    Point p = new Point(  2.0, 3.0 );
    Point q = new Point3( 2.0, 3.0, 4.0 );
    ((Point3)q).up( 0.7 ); System.out.println("q = " + q);
    ((Point3)p).up( 0.7 ); System.out.println("p = " + p);
}}
```

```
q = (2.0, 3.0, 4.7)
java.lang.ClassCastException: Point
        at DownCastError.main(DownCastError.java:18)
```

Static versus Dynamic Typing

- Compile-time type checking: **Static typing**
- Run-time type checking: **Dynamic typing**
- All type errors will be ***detected***: **strongly typed** languages
- A program is **type safe** if it is known to be free of type errors
- A language is **type safe** if all its programs are type safe

Warning: *Quite some mix-up in the conclusions in the textbook!*

- For every language, all its programs pass all its statical tests:
  - For a dynamically typed language like LISP, at least some programs contain type errors, otherwise no dynamic checking would be necessary: LISP may be strongly typed, but is **not type safe**. (p. 51)
  - Java is strongly typed, but in some aspects **only with dynamic type checks**: Java is **not type safe!** (p. 233)
  - **Haskell is strongly statically typed**, and therefore **type safe!** (p. 233)

Oberon type guards correspond to Java down-casts.

Type Languages

At each point in a program, there is a **type language** $\mathcal{T}$

- most languages include implementation-oriented **primitive types** like `int`, `bool`, `float`, `char`
- $\mathcal{T}_{\text{Jay}} = \{\texttt{int}, \texttt{boolean}\}$
- **type definitions** extend the type language
- **type constructors** produce infinite type languages:
  - **Oberon:** `ARRAY` *N* `OF`, `POINTER TO` only
  - **C, Java:** `*`, `[]` only
  - **Haskell:** `_->_ , [_] , (_,_) , Maybe _ , IO _ , Ratio _ , Array _ _` , and **user-defined type constructors**, e.g.: `FiniteMap _ _ , Set _ , Graph _`
  - **Java 1.5**: Will have "**generics**", i.e., parametric polymorphism similar to Haskell

Type System Principles

Assume a constant type language $\mathcal{T}$.

- At each point in a program, there is a **type context** (or **typing environment**) Γ, mapping visible identifiers to types.

  Usually, this is a (finite) partial function:

  $$\Gamma \;:\; \mathit{Identifier} \mathrel{+\!\!\!\!\rightarrow} \mathcal{T}$$

- Given a type context Γ,
  - a **declaration**, if **valid**, produces a new type context Γ'
  - an **expression** *e* may **have a type** *t*

    $$\Gamma \;\vdash\; e : t$$

    (The, *e* is **well-typed**, **with type** *t*)
  - a **statement** may **be well-typed**

Jay: Extracting the Context from the Declarations

Given a type context Γ, a **declaration**, if **valid**, produces a new type context Γ'.

- Abstract syntax: **class** *Declaration* { *Variable v*; *Type t*; }
  class *Declarations* **extends** *Vector* {}
- Java type *TypeMap* implements $\mathit{Identifier} \mathrel{+\!\!\!\!\rightarrow} \mathcal{T}$
- Jay has only one declaration block: can start from empty context:

  $$\begin{aligned} &\mathit{typing} \;:\; \mathit{Declarations} \rightarrow \mathit{TypeMap} \\ &\mathit{typing}\ (\mathit{Declarations}\ d) = \bigcup_{i \in \{1,\ldots,n\}} \{d_i.v \mapsto d_i.t\} \end{aligned}$$

-
```
TypeMap typing (Declarations d) {
    TypeMap map = new TypeMap();
    for (int i=0; i<d.size(); i++)
        map.put (((Declaration)(d.elementAt(i))).v,
                 ((Declaration)(d.elementAt(i))).t);
    return map;
}
```

Practice!

- Exercises 3.1 – 3.3
- Add error messages to validity and type checking functions
- Test with correct and incorrect Jay programs

Jay: Checking Validity of Declarations

- **Overloaded validity function** *V*
- **Validity of declaration block**: Each variable name declared at most once:

$$V \;:\; Declarations \to \mathbf{B}$$
$$V\,(Declarations d) \;=\; \forall i, j : \{1, \ldots, n\} \bullet (i \neq j \Rightarrow d_i.v \neq d_j.v)$$

- Implementation:

```
public boolean V (Declarations d) {
  for (int i=0; i<d.size() – 1; i++)
    for (int j=i+1; j<d.size(); j++)
      if ((((Declaration)(d.elementAt(i))).v).equals
            (((Declaration)(d.elementAt(j))).v))
        return false;
  return true;
}
```

Jay Expression Typing Rules

```
class Expression {} // Expression = Variable | Value | Binary | Unary
class Variable extends Expression { String id; }
class Value extends Expression { // Value = int intValue | boolean boolValue
  Type type; int intValue; boolean boolValue; }
class Binary extends Expression {
  Operator op; Expression term1, term2; }
class Operator { String val; }
```

- Variables must have been declared with of the two types `int` and `boolean`
- Arithmetic operators `+`, `-`, `*`, `/` demand two `int` arguments and produce an `int` expression
- Relational operators `==`, `!=`, `<`, `<=`, `>`, `>=` demand two `int` arguments and produce a `boolean` expression
- Boolean operators `&&`, `||` demand two `boolean` arguments and produce a `boolean` expression

---

Jay Expression Type Inference

```
public Type typeOf (Expression e, TypeMap tm) {
  if (e instanceof Value) return ((Value)e).type;
  if (e instanceof Variable) {
    Variable v = (Variable)e;
    if (!tm.containsKey(v)) return new Type(Type.UNDEFINED);
    else return (Type) tm.get(v);                          }
  if (e instanceof Binary) {
    Binary b = (Binary)e;
    if (b.op.ArithmeticOp( )) return new Type(Type.INTEGER);
    if (b.op.RelationalOp( ) || b.op.BooleanOp( ))
                          return new Type(Type.BOOLEAN);
  }
  if (e instanceof Unary) {
    Unary u = (Unary)e;
    if (u.op.UnaryOp( )) return new Type(Type.BOOLEAN);  }
  return null;
}
```

Only inspects top-level construction!

Jay Expression Type Checking

```
public boolean V (Expression e, TypeMap tm) {
  if (e instanceof Value) { return true; }
  if (e instanceof Variable) { return tm.containsKey((Variable)e); }
  if (e instanceof Binary) {
    Type typ1 = typeOf(((Binary)e).term1, tm);
    Type typ2 = typeOf(((Binary)e).term2, tm);
    if (! V (((Binary)e).term1, tm)) return false;
    if (! V (((Binary)e).term2, tm)) return false;
    if (((Binary)e).op.ArithmeticOp( ) || ((Binary)e).op.RelationalOp( ))
      return typ1.isInteger() && typ2.isInteger();
    if (((Binary)e).op.BooleanOp( ))
      return typ1.isBoolean() && typ2.isBoolean();              }
  if (e instanceof Unary) {
    Type typ1 = typeOf(((Unary)e).term, tm);
    return typ1.isBoolean() && V(((Unary)e).term, tm)
                        && (((Unary)e).op.val).equals("!") ;      }
  return false;
}
```