Bounded Polymorphism

In Pizza we can use interfaces to bound the type parameters a class will accept.

Recall our **`Compare`** interface:

```
interface Compare {
 boolean lessThan(Object o1,
                  Object o2);
}
```

We can specify that a parameterized class will only takes types that implement **`Compare`**:

```
class LinkedList<T implements
                 Compare> { ... }
```

In fact, we can improve upon how interfaces are defined and used.

Recall that in method **`lessThan`** we had to use parameters declared as type **`Object`** to be general enough to match (and accept) any object type. This leads to clumsy casting (with run-time correctness checks) when **`lessThan`** is implemented for a particular type:

```
class IntCompare implements Compare {
 public boolean lessThan(Object i1,
                         Object i2){
 return ((Integer)i1).intValue() <
        ((Integer)i2).intValue();}
}
```

Pizza allows us to parameterize class definitions with type parameters, so why not do the same for interfaces?

In fact, this is just what Pizza does. We can now define **`Compare`** as

```
interface Compare<T> {
  boolean lessThan(T o1, T o2);
}
```

Now we define class **`LinkedList`** as

```
class LinkedList<T implements
            Compare<T> > { ... }
```

Given this form of interface definition, no casting (from type **`Object`**) is needed in classes that implement **`Compare`**:

```
class IntCompare implements
   Compare<Integer> {
 public boolean lessThan(Integer i1,
                         Integer i2){
 return i1.intValue() <
        i2.intValue();}
}
```

First-class Functions in Pizza

In Java, functions are treated as constants that may appear only in classes.

To pass a function as a parameter, you must pass a class that contains that function as a member. For example,

```
class Fct {
 int f(int i) { return i+1; }
}
class Test {
  static int call(Fct g, int arg)
    { return g.f(arg); }
}
```

Changing the value of a function is even nastier. Since you can't assign to a member function, you have to use subclassing to override an existing definition:

```
class Fct2 extends Fct {
 int f(int i) { return i+111; }
}
```

Computing new functions during executions is nastier still, as Java doesn't have any notion of a lambda-term (that builds a new function).

Pizza makes functions first-class, as in ML. You can have function parameters, variables and return values. You can also define new functions within a method.

The notation used to define the type of a function value is

$(T_1,T_2, \ldots)\text{->}T_0$

This says the function will take the list $(T_1,T_2, \ldots)$ as it arguments and will return T_0 as its result.

Thus

```
(int)->int
```

represents the type of a method like

```
int plus1(int i) {return i+1;}
```

The notation used by Java for fixed functions still works. Thus

```
static int f(int i){return 2*i;};
```

denotes a function constant, **f**.

The definition

```
 static (int)->int g = f;
```

defines a field of type **(int)->int** named **g** that is initialized to the value of **f**.

The definition

```
static int call((int)->int f,
                int i)
    {return f(i);};
```

defines a constant function that takes as parameters a function value of type **(int)->int** and an **int** value. It calls the function parameter with the **int** parameter and returns the value the function computes.

Pizza also has a notation for anonymous functions (function literals), similar to **`fn`** in ML and **`lambda`** in Scheme. The notation

**`fun (`T_1 a_1`, `T_2 a_2`, ...) -> `T_0
`  {Body}`**

defines a nameless function with arguments declared as **`(`T_1 a_1`, `T_2 a_2`, ...)`** and a result type of **T_0**. The function's body is computed by executing the block **`{Body}`**.

For example,

```
static (int)->int compose(
  (int)->int f, (int)->int g){
   return fun (int i) -> int
    {return f(g(i));};
}
```

defines a method named **`compose`**. It takes as parameters two functions, **`f`** and **`g`**, each of type **`(int)->int`**.

The function returns a function as its result. The type of the result is **`(int)->int`** and its value is the composition of functions **`f`** and **`g`**:

```
  return f(g(i));
```

Thus we can now have a call like

```
compose(f1,f2)(100)
```

which **computes `f1(f2(100))`**.

With function parameters, some familiar functions can be readily programmed:

```
class Map {
 static int[] map((int)->int f,
                  int[] a){
   int [] ans =
       new int[a.length];
   for (int i=0;i<a.length;i++)
       ans[i]=f(a[i]);
   return ans;
 };
}
```

And we can make such operations polymorphic by using parametric polymorphism:

```
class Map<T> {
  private static T dummy;
  Map(T val) {dummy=val;};
  static T[] map((T)->T f,
                 T[] a){
     T [] ans = (T[]) a.clone();
     for (int i=0;i<a.length;i++)
         ans[i]=f(a[i]);
     return ans;
  };
}
```

Algebraic Data Types

Pizza also provides “algebraic data types” which allow a type to be defined as a number of cases. This is essentially the pattern-oriented approach we saw in ML.

A list is a good example of the utility of algebraic data types. Lists come in two forms, null and non-null, and we must constantly ask which form of list we currently have. With patterns, the need to consider both forms is enforced, leading to a more reliable programming style.

In Pizza, patterns are modeled as “cases” and grafted onto the existing switch statement (this formulation is a bit clumsy):

```
class List {
  case Nil;
  case Cons(char head,
            List tail);
  int length(){
    switch(this){
      case Nil: return 0;
      case Cons(char x, List t):
          return 1 + t.length();
    }
  }
}
```

And guess what! We can use parametric polymorphism along with algebraic data types:

```
class List<T> {
  case Nil;
  case Cons(T head,
            List<T> tail);
  int length(){
    switch(this){
     case Nil: return 0;
     case Cons(T x, List<T> t):
        return 1 + t.length();
    }
  }
}
```

Enumerations as Algebraic Data Types

We can use algebraic data types to obtain a construct missing from Java and Pizza—enumerations.

We simply define an algebraic data type whose constructors are not parameterized:

```
class Color {
  case Red;
  case Blue;
  case Green;
  String toString() {
    switch(this) {
     case Red: return "red";
     case Blue: return "blue";
     case Green: return "green";
    }
  }
}
```

This approach is better than simply defining enumeration values as constant (final) integers:

```
final int Red = 1;
final int Blue = 2;
final int Green = 3;
```

With the algebraic data type approach, **`Red`**, **`Blue`** and **`Green`**, are not integers. They are constructors for the type **`Color`**. This leads to more thorough type checking.

Reading Assignment

- Python Tutorial
  (linked from class web page)

Python

One of the newest and most innovative scripting languages is *Python*, developed by Guido van Rossum in the mid-90s. Python is named after the BBC "Monty Python" television series.

Python blends the expressive power and flexibility of earlier scripting languages with the power of object-oriented programming languages.

It offers a lot to programmers:

- An interactive development mode as well as an executable "batch" mode for completed programs.
- Very reasonable execution speed. Like Java, Python programs are compiled. Also like Java, the compiled code is in

an intermediate language for which an interpreter is written. Like Java this insulates Python from many of the vagaries of the actual machines on which it runs, giving it portability of an equivalent level to that of Java. Unlike Java, Python retains the interactivity for which interpreters are highly prized.

- Python programs require no compilation or linking. Nevertheless, the semi-compiled Python program still runs much faster than its traditionally interpreted rivals such as the shells, *awk* and *perl*.
- Python is freely available on almost all platforms and operating systems (Unix, Linux, Windows, MacOs, etc.)

- Python is completely *object oriented*. It comes with a full set of objected oriented features.
- Python presents a first class object model with first class functions and multiple inheritance. Also included are classes, modules, exceptions and late (run-time) binding.
- Python allows a clean and open program layout. Python code is less cluttered with the syntactic “noise” of declarations and scope definitions. Scope in a Python program is defined by the *indentation* of the code in question. Python thus breaks with current language designs in that white space has now once again acquired significance.

- Like Java, Python offers automated memory management through run-time reference counting and garbage collection of unreferenced objects.
- Python can be embedded in other products and programs as a control language.
- Python's interface is well exposed and is reasonably small and simple.
- Python's license is truly public. Python programs can be used or sold without copyright restrictions.
- Python is extendable. You can dynamically load compiled Python, Python source, or even dynamically load new machine (object) code to provide new features and new facilities.

- Python allows low-level access to its interpreter. It exposes its internal plumbing to a significant degree to allow programs to make use of the way the plumbing works.
- Python has a rich set of external library services available. This includes, network services, a GUI API (based on tcl/Tk), Web support for the generation of HTML and the CGI interfaces, direct access to databases, etc.

Using Python

Python may be used in either interactive or batch mode.

In interactive mode you start up the Python interpreter and enter executable statements. Just naming a variable (a trivial expression) evaluates it and echoes its value.

For example (>>> is the Python interactive prompt):

```
>>> 1
1
>>> a=1
>>> a
1
>>> b=2.5
>>> b
2.5
```

```
>>> a+b
3.5
>>> print a+b
3.5
```

You can also incorporate Python statements into a file and execute them in batch mode. One way to do this is to enter the command

`python file.py`

where **`file.py`** contains the Python code you want executed. Be careful though; in batch mode you must use a **`print`** (or some other output statement) to force output to be printed. Thus

```
1
a=1
a
```

```
b=2.5
b
a+b
print a+b
```

when run in batch mode prints only **`3.5`** (the output of the **`print`** statement).

You can also run Python programs as Unix shell scripts by adding the line **`#! /usr/bin/env python`** to the head of your Python file.

(Since # begins Python comments, you can also feed the same augmented file directly to the Python interpreter)

Python Command Format

In Python, individual primitive commands and expressions must appear on a single line.

This means that

```
a = 1
+b
```

does not assign `1+b` to `a`! Rather, it assigns `1` to `a`, then evaluates `+b`.

If you wish to span more than one line, you must use `\` to escape the line:

```
a = 1     \
  +b
```

is equivalent to

```
a = 1 +b
```

Compound statements, like **`if`** statements and **`while`** loops, *can* span multiple lines, but individual statements within an **`if`** or **`while`** (if they are primitive) must appear one a single line.

Why this restriction?

With it, `;`'s are mostly unnecessary!

A `;` at the end of a statement is legal but usually unnecessary, as the end-of-line forces the statement to end.

You can use a `;` to squeeze more than one statement onto a line, if you wish:

```
a=1; b=2 ; c=3
```

Identifiers and Reserved Words

Identifiers look much the same as in most programming languages. They are composed of letters, digits and underscores. Identifiers must begin with a letter or underscore. Case is significant. As in C and C++, identifiers that begin with an underscore often have special meaning.

Python contains a fairly typical set of reserved words:

```
and       del       for     is      raise
assert    elif      from    lambda  return
break     else      global  not     try
class     except    if      or      while
continue  exec      import  pass
def       finally   in      print
```

Numeric Types

There are four numeric types:

1. Integers, represented as a 32 bit (or longer) quantity. Digits sequences (possibly) signed are integer literals:
   `1  -123  +456`
2. Long integers, of unlimited precision. An integer literal followed by an **`l`** or **`L`** is a long integer literal:
   `123456789000000000000000L`
3. Floating point values, represented as a 64 bit floating point number. Literals are of fixed decimal or exponential form:
   `123.456  1e10  6.0231023`
4. Complex numbers, represented as a pair of floating point numbers. In complex literals **`j`** or **`J`** is used to

denote the imaginary part of the complex value:

```
1.0+2.0j  -22.1j  10e10J+20.0
```

There is no character type. A literal like `'a'` or `"c"` denotes a string of length one.

There is no boolean type. A zero numeric value (any form), or **`None`** (the equivalent of void) or an empty string, list, tuple or dictionary is treated as false; other values are treated as true.

Hence

```
"abc" and "def"
```

is treated as true in an **`if`**, since both strings are non-empty.