Language Paradigms

Different ways of expressing computation;

- imperative (How you program)
- functional (How you "think" about problems*)

Others:

object-oriented, logical, dataflow, coordination, algebraic, graph-based, etc.

Note: distinction is sometimes fuzzy!

* – Mostly! Certain problems more naturally lend themselves to certain paradigms

Imperative Paradigm

Procter
from Amtoft
from Hatcliff
from Leavens

Example: compute m^n $(n \geq 0)$

```
result = 1;
while(n > 0){
    result = result * m;
    n = n - 1;
}
```

Assessment:

- computation is expressed by repeated modification of an *implicit* store (i.e., components *command* a store modification),
- intermediate results are held in store
- iteration (loop)-based control

Functional Paradigm

Example: compute m^n $(n \geq 0)$

```
fun power (m,n) =
    if (n = 0)
       then 1
       else m * power(m, n - 1);
```

Assessment:

- computation is expressed by function application and composition
- no implicit store
- intermediate results (function outputs) are passed directly into other functions
- recursion-based control

A Guide Rope

Procter
from Amtoft
from Hatcliff
from Leavens

"When people talk about functional programming, they mention a dizzying number of 'functional' characteristics. They mention immutable data, first class functions and tail call optimisation. These are language features that aid functional programming. They mention mapping, reducing, pipelining, recursing, currying and the use of higher order functions. These are programming techniques used to write functional code. They mention parallelization, lazy evaluation and determinism. These are advantageous properties of functional programs.

Ignore all that. Functional code is characterised by one thing: the absence of side effects. It doesn't rely on data outside the current function, and it doesn't change data that exists outside the current function. Every other 'functional' thing can be derived from this property. Use it as a guide rope as you learn."

– Mary Rose Cook, A Practical Introduction to Functional Programming

Why Functional Programming?

- **Advanced features:** Many of the features of functional languages, such as parametric polymorphism, pattern matching, and advanced modules are very elegant and either do not appear in other languages like Java, C++, etc., or are much less elegant (Java's Generics, C++ Templates).
- **Very high-level:** Using SML lets us describe language processors very succinctly (much more concisely than any imperative language).
- **Clean:** Functional languages is useful for various critical applications where programs need to be proven correct.
- **It's not Java:** At some point in your career, you will have to learn a new language. This course prepares you for that by asking you to learn a new, radically different language quickly. This forces you to think more deeply (mental pushups!).

Why SML?

- **Well-understood foundations:** This is a course about the foundations of programming languages, and the theory/foundations of SML have been studied more in recent years than almost any other language.
- **Well-designed:** Robin Milner, the principal designer of SML received the Turing Award, in part, because of his work on SML.
- **There's more!** There are also several different concurrent versions of SML, object-oriented extensions, libraries for various applications,

Statements vs. Expressions

- Statement: What code does
- Expression: What code is

This closely mirrors the difference between imperative and functional languages!

- Imperative: A language that emphasizes statements
- Functional: A language that emphasizes expressions

– Gabriel Gonzalez, Haskell For All

Statements

- What code does, or
- Construct evaluated only for its effect

Examples:

```
int elements[5] = {1, 2, 3, 4, 5};
int total = 5 - 5;

for(int i=3*0; i < 5; i+=getOne()){
    total = total + i;
}

System.out.println(total);
```

Statements

- What code does, or
- Construct evaluated only for its effect

Examples:

```
int elements[5] = {1, 2, 3, 4, 5};
int total = 5 - 5;

for(int i=3*0; i < 5; i+=getOne()){
    total = total + i;
}

System.out.println(total);
```

Statement-oriented/imperative languages:

- Pascal, C, C++, Ada, FORTRAN, COBOL, etc.

Expressions

- What code is, or
- construct evaluated to yield value

Examples:

```
int elements[5] = {1, 2, 3, 4, 5};
int total = 5 - 5;

for(int i=3*0; i < 5; i+=getOne()){
    total = total + i;
}

System.out.println(total);
```

Expressions

- What code is, or
- construct evaluated to yield value

Examples:

```
int elements[5] = {1, 2, 3, 4, 5};
int total = 5 - 5;

for(int i=3*0; i < 5; i+=getOne()){
    total = total + i;
}

System.out.println(total);
```

Pure expressions: no side-effects

Expression-oriented/functional languages:

- Scheme, ML, Lisp, Haskell, Miranda, FP, etc

Statements vs. Expressions

If it's unclear...

- Remember the guide rope! Ask yourself, "is this construct evaluated for its value, or its side-effect?"

As always, MSDN documentation is stellar

- Statements (C# Programming Guide)
- Expressions (C# Programming Guide)