AFL: Another Fun Language

AFL is a small simple programming language, designed to illustrate and to facilitate experimentation with some of the basic principles of programming languages. The language supports a wide range of control structures and rich operator definition capabilities, based on a small core set of features, and provides access to the mechanics of the language. Most interestingly, it supports "true" generators, pattern matching, coroutines, in-language pipes, and continuations.

The long-term goal of the AFL project is to explore and illustrate what's needed to improve the support for text and markup processing in existing and new programming languages. The current AFL language only partially covers what's required. It's primary focus is on processing forms — on language structures rather than on data structures. Even so, it's already a rich language, capable of many things that you can't do in most programming languages. The next phase of development will focus on type system and data organization issues.

The current AFL language is fully implemented and documented. It's description and a download can be found at:

www.wilmott.ca/afl

Using AFL: Coroutines

Using Symmetric Function Calls:

```
def cr1 (suspend) :
 {
  print "Start cr1";
  for 1 to 5 do (n):
   (
    print "cr1 passes to cr2: " ++ n;
    print "cr1 resumed: " ++ suspend (n);
   );
  print "cr1 exits";
 };

def cr2 (suspend, firstArg) :
 {
  print "Start cr2 with: " ++ firstArg;
  for 5 by -1 do (n):
   (
    print "cr2 passes to cr1: " ++ n
   print "cr2 resumed: " ++ suspend (n);
   );  # cr2 never exits
 };

cr1 (makecoroutine (cr2, done));
```

The Program's Output:

```
Start cr1
cr1 passes to cr2: 1
Start cr2 with: 1
cr2 passes to cr1: 5
cr1 resumed: 5
cr1 passes to cr2: 2
cr2 resumed: 2
cr2 passes to cr1: 4
cr1 resumed: 4
cr1 passes to cr2: 3
cr2 resumed: 3
cr2 passes to cr1: 3
cr1 resumed: 3
cr1 passes to cr2: 4
cr2 resumed: 4
cr2 passes to cr1: 2
cr1 resumed: 2
cr1 passes to cr2: 5
cr2 resumed: 5
cr2 passes to cr1: 1
cr1 resumed: 1
cr1 exits
```

Using AFL: More Coroutines

Piped Writing and Pattern Matching

```
def LETTER: "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
            "abcdefghijklmnopqrstuvwxyz";

def capitalize []:
  loop (exit):
    if ~: noneOf LETTER :+ then
      << matched []
    else if ~: anyOf LETTER then
      << toUpper matched []
      << +: anyOf LETTER :*
    else
      exit '{};

<< (stringReader ("hello world!") <~
    capitalize).readtoend ();
```

The Program's Output:

```
Hello World!
```

"capitalize" is a "filter", a function that converts its input into its output. <~ feeds its first argument to its second.

True Generators Are Coroutines

AFL's iterators and generators are based on program state rather than data state, as in Java, C# and Python:

```
def [136 a] by [137 s] to [136 z]
    (yield): {
  def n = a;
  while *n < z do {
    yield (*n);
    n += s;
  }};
```

The difference between AFL and the other languages is that "yield" can be passed around arbitrarily. For example, you can "yield" an XML markup event from deep inside an XML parser, making "pull" parsers much easier to implement. (The outer "def" above is an operator definition for "by-to".)

Programming v.s Templating Languages

Hello World Revisited:

The venerable "Hello World!" program can be written in AFL in a number of ways:

1. A program consisting entirely of:

```
"Hello World!"
```

returns "Hello World!" (sans quotes) as its result to the system invoking the program, which then displays it.

2. You can go, more traditionally:

```
print "Hello World!";
```

3. You can set /k=t (make AFL a template language!) on the AFL compiler's command line and have your program be:

```
Hello World!
```

More About Templating:

The template version of the "Hello World!" program is the simplest template program: just text. The next step is to interpolate values:

```
\{def what: "World"}Hello \{what}!
```

The difference between "regular" and template languages is whether the text in a program needs quoting or whether the code needs quoting: AFL has both.

A further refinement is XML templating:

```
print <book><title>\{title}</title>
        <author>\{author}</author></book>
```

The point being that what we need is not different languages, but different ways of expressing languages.