Yet Another Programming Language

Sam Wilmott

24 February 2004

Too Many Programming Languages

- There are lots and lots of programming languages, most of the commonly used ones very similar to each other, bar the odd feature and syntactic nicety. And there's no good (technical) reason for this plethora.
- There are also a lot of specialized programming languages, many that should be better known, with useful stuff in them.

Why Yet Another Programming Language?

- So with too many programming languages already, how can one justify yet another? Well:
  - There are important features missing from all commonly used languages.
  - Most programming languages are too much of a compromise with ancient hardware limitations.

Why Yet Another Programming Language?

- Many interesting and potentially useful features are hidden in the midst of otherwise gratuitously different syntax, making them inaccessible to most programmers.
- Most programmers are still stuck with what are essentially low-level programming languages: C++, C#, Java, Perl, Python.

Why Yet Another Programming Language?

- Maybe most importantly, there hasn't been another major Pascal language since the '70's – even Pascal isn't a "Pascal" anymore.
- Pascal was originally intended for teaching – not just for teaching programming but for teaching how programming language are designed and implemented, and how much can be done with a small language.

Why Not Just Talk About What's Needed?

- Mostly, because there's been a lot of talking over the last four decades, and talking has been proven to not be enough.
- Most of us need to see and touch to understand – we learn by doing.
- Most programming languages are implemented in a way that makes it expensive to play with their designs.

So What's Missing?

- "Threads", a.k.a. asynchronous threading, has been all the rage for a long time now, but their equally useful synchronous cousin, coroutines, a.k.a. synchronous threading, are rarely to be seen.
- True generators, as opposed to the "simple" form in C# and Python, are a key feature of any text processing applications.

So What's Missing?

- Properly implemented generic types have taken far too long to get into major languages – C# cries out for them, and they are just arriving.
- Object Oriented methodology is usually just the thing type-system-wise, but sometimes it's a limitation. After-the-fact type manipulation facilities, like unions, would often help.

So What's Needed?

- Lots of things – we are still in the early days of programming language development.
- I have no idea what the future of programming languages is, or what future programming languages will look like.
- But I do know that there's a lot yet to do – in the face of massive conservatism in the programmer community.

So What's Needed?

- One thing that I think is needed is a new "teaching" language – that serves in the early 21st century what Pascal did in the '70's.
- Such a programming language would be used to demonstrate and to explore basic programming language concepts and functionality, and to provide the basis for a discussion on where we should be going.

What Would A New Language Look Like?

- It would have a familiar syntax (sorry Lisp) without being overly constrained by the conventions of current languages (sorry C).
- It would incorporate functional and syntactic forms from the lowest level to the highest. With, as much as possible, the higher-level forms described in terms of the lower.
- It would be small. Small is still beautiful.

AFL – A New Programming Language

AFL

- AFL: “Another Fun Language”.
- It has been said that the appeal of the Python programming language is that it’s fun.
- The intent is that AFL is likewise fun, if in a different way.

AFL

- AFL's design is based on a number of principles:
  - Concede nothing to current hardware design or to performance limitations.
  - Keep the language as small as possible.
  - Implement as much, both functionally and syntacticly, in the language itself.

AFL

- Probably the most important feature of AFL from a functional point of view is that it has no "stack frame" – everything goes on the "heap".
- Everything, including argument lists and local variables, are allocated so as to be potentially persistent – there's no *a priori* assumption as to the use of any set of values or how any name will be later referred to.

AFL

- AFL is a “prototype”, in its design, implementation and intended use.
- All performance issues are being postponed until the language design settles.
- AFL’s design, implementation, documentation and use are developing in parallel: features can be added, tested and removed quickly.

AFL

- AFL is being developed in stages.
  - Stage One is functionality.
  - Stage Two is a type system, including but not limited to an object-based models (actually more than one model).
  - Stage Three is optimization – improving performance as required to make the language usable.

AFL and .NET

- AFL is implemented using the .NET platform. .NET provides:
  - a garbage collector,
  - interesting and useful “machine” instructions, and
  - a rich and easily accessible run-time library.

Where Things Stand

- Work on AFL started in October 2004.
- Implementing AFL isn't the only thing I've been doing: there's been a lot of design and experimentation.
- I've been learning too. AFL is my learning tool for C# and .NET – the AFL compiler is my first C# and .NET program.

Where Things Stand

- The current (Stage One) iteration of AFL:
  - is an Algol-like dynamically typed language,
  - with a compiler written in C# that translates AFL programs to C# or to CIL assembler language (which then can be compiled to runnable programs), and
  - with an overview document describing the language.

Where Things Stand

- The current implementation of AFL has arithmetic, string and logical operations, records, hash tables, dynamic vectors, first-class functions, user-defined operators (multi-argument, prefix, infix and suffix), coroutines, true generators, “if” and looping forms, optional template programming, asynchronous threading, pattern matching, text file I/O and an XML parser interface.

Where Things Are Going

- Stage Two is primarily about adding a type system, including type definitions, a type algebra, generic types, interfaces, objects and classes, with multiple object models, supporting both dynamic and static typing.
- As with Stage One, the emphasis will be on defining as as much as possible in the language itself, based on a minimal core language.

Where Things Are Going

- Besides the basics, work on ongoing improvements in the implementation:
  - Create .NET assemblies directly from the AFL compiler: both compile-and-go and runnable .exe files.
  - More meaningful error reporting at run time is badly needed.
- And more documentation and a tutorial.

More About AFL

www.wilmott.ca/afl

Downloads, updates and news.