Yet Another Programming Language

Sam Wilmott
420 Tweedsmuir Ave, Ottawa, ON, Canada
sam@wilmott.ca

Abstract

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. The language's initial implementation exploits the high speed of currently available computers, focusing on functionality rather than performance.

This is an explanatory overview of the current state of AFL. It's neither intended to be a language specification nor a tutorial.

1 Introduction

1.1 Why Another Programming Language?

First and foremost, AFL is a learning exercise. I've been learning while developing it, and I hope that others can learn from what I've done.

The current AFL language is intended to form the foundation for further development. In fact, my original intention in developing AFL was to play with type system ideas, but it turned out that doing that required a functional foundation on which to build. Hence the current language.

Where AFL might best fit in the programming language world is in a teaching role: helping future programming language designers understand and experiment with different forms of control structures. There's nothing in AFL that's new or unique – it's all been done before – but even so, much of it is not as familiar as it should be, and AFL attempts to make the mechanics of the language more accessible.

From a use point of view, where the current AFL language differs mostly from commonly used programming languages is in its support of a wide range of process or control-flow mechanisms. After all, programming is about data and about process. But most of the advances in commonly used programming languages have been in the area of data structure. The opportunities for further major advances are primarily in the area of process and control structures.

1.2 Principles

The AFL languages are being developed based on a number of principles:

- Make the AFL languages be "computing languages" rather than "computer languages". As few features of the languages as are practical should be based on current computer hardware constraints.

  A lot of what's in commonly used programming languages is there as a compromise with hardware limitations. Such features limit what programmers can do and how we do it. Major limiting features of current programming languages, such as having a single function call stack, date from the 1960's and should be obsolete by now. The fault is not just with programming language design – current operating system and CPU designs seem to be optimized to the needs of older-style programming languages and programming styles.
- Make the language as easy to follow as possible. Stick with familiar language forms where they fit, but don't be afraid of less familiar syntax where it easies understanding what's going on.
- Make the implementation as flexible as possible. The language is easy to modify to illustrate different approaches.
- As much as possible, make low-level functionality available at a high level. This maximizes what can be implemented using the language itself. More importantly, it makes it possible to explain what's going on "under the hood" in terms of the language itself rather than by dropping into an explanation of computer architecture.
- Avoid any "X-oriented" approach. Although there's a general tendency to think of things like "object-oriented", "functional programming" or "text processing" as features of programming languages, they are actually programming models. AFL should, as much as possible, support different programming models without being specifically oriented one way or the other.

  That said, the reader will no doubt notice a distinct bias in favour of text processing in the facilities implemented in AFL and its include files. That's based on the uses I've made of the language so far.
- Incorporate both arcane and familiar aspects of programming languages.

  One of the chief difficulties with currently available programming languages is a tendency to assume that the core language has a single audience – that all programmers, at all times, use

the same mechanisms. But one size does not fit all. There are end-user application developers and core functionality library developers, and their needs overlap but are not the same – many programmers fulfill different roles and have different needs at different times.

- Develop the AFL language design, implementation and documentation in parallel – and don't be commit to any of the three until all three stabilize. This way, different approaches have been experimented with, simplifications were made when discovered, and focus has be placed on the easiest way to explain what's going on. This approach also makes the language as self-consistent as possible, which aids in understanding and use.
- Make the language as "extensible" as possible. Allow users to implement their own operators and control structures. And as much as possible in the language itself, rather than having to fiddle with the underlying implementation.
- Avoid addressing performance issues early on. As said, this is a computing language. This approach has two consequences: it's made the language easy to develop and experiment with, and it's made the performance of the language very slow. The later issue has been dealt with – to some extent at least – by investing a few thousand extra dollars in fast hardware, and has paid off well in facilitating experimentation.

  There's no reason for the performance to remain so poor. An optimization pass or two can improve the performance considerably, as can removing some of the low-level features that are useful in their explanatory role, but not of much use to real-world programmers.

1.3 Why "AFL"?

"AFL" stands for "Another Fun Language". (Or for those of you totally fed up with the proliferation of programming languages, "Another (something else) Language".) One of the chief stated appeals of the Python programming language has been that "it's fun to program in". I'm hoping that AFL is fun too, both because that makes it easier to learn, and because I like having fun programming.

The AFL languages are built in layers. At present, the following languages exist:

- AFL0 ("eh-ef-el-zero") implements the basic core functionality of all the AFL languages. It's purposely small. Everything outside of AFL0 is first translated into AFL0, before being compiled into a lower-level form.
- AFL1 ("eh-ef-el-one") implements basic syntactic and definitional forms such as functions, operators and local scopes, simplifies the name model and manages intermediate values.
- AFL2 ("eh-ef-el-two") consists entirely of a library of functionality written in AFL itself. It implements things considered basic in most programming languages such as if-then-else, loops, coroutines, variables and array data structures.

The "0" in AFL0 is because it's the starting point. It's the foundation for further work. The name "AFL" is used to cover all the AFL languages. Where it's used, what's said applies to all the languages.

1.4 Implementation

AFL has been developed using Microsoft's .NET Framework under Windows XP, and using the Mono platform under Mac OS 10.3 and 10.4. I've run it under Windows (2000 and XP), MacOS (10.3, Panther and 10.4, Tiger) and Linux (SUSE 9.1).

AFL is implemented using a compiler that produces run-time code. There's no interpreter. At present, the compiler targets are C# source code and CIL (.NET assembler language) source code, both of which require a further step of compilation to .NET ".exe" assemblies.

Why .NET? you might ask. For a number of reasons:

- To determine whether and to demonstrate that such a language can be implemented.
- I wanted to learn C# and learn about the .NET run-time system. Developing AFL has been my learning tool. (The AFL implementation is my first C# project.)
- .NET has a garbage collector and a rich library of functionality, so I don't have to bother implementing that stuff myself.
- The .NET run-time checks running code for type correctness, which is a substantial aid in programming language development.
- The .NET "virtual machine" has a tail-calling instruction in the run-time code. This feature has been used extensively in the AFL implementation of control flow structures.
- For portability: .NET and hence AFL runs on a variety of platforms, thanks to the Mono Project, Microsoft's Shared Source CLI and the Gnu dotnet implementation.
- To explore, illustrate and push the limitations of current hardware and "VM" (Java and .NET) designs.

The work could have been done using the Java language and its run-time (JVM), or sticking to a specific computer architecture (e.g. Intel or PowerPC). The choice of .NET over the JVM was motivated by my wanting to learn something new, the tail-calling instruction, and the multi-language support of .NET.

1.5 Difficulties

There's a number of problems with the current implementation. Primarily, because of the highly dynamic nature of the language, many errors don't show up at compile-time, and produce hard-to-interpret results at run time. In part this is a drawback of any dynamically typed language, and the difficulty is compounded by AFL's "stacklessness" – there's no call stack, so it's difficult to determine the context of an error.

It isn't all dark however – the .NET run-time does catch most type errors. As well, the AFL implementation includes tracing facilities and AFL-specific run-time checks. Further work could improve the situation, for example determining a problem's context in common use cases.

The other difficulty with the current implementation is that it's amazingly slow – it's not really usable other than for small examples at present. No attempt is made to optimize the code, even in the simplest and most obvious way. Again, a major improvement in this area could be achieved with the introduction of some basic optimization. It'll never be a really fast implementation though: the

availability of continuations in all contexts makes some important optimizations difficult.

1.6 Where AFL Stands And Where It's Headed

As a family of languages, AFL is a work in progress. It's best to think of the current generation of AFL languages and their implementation as a prototype. The syntax currently used is suitable for explaining ideas, but not necessarily the best for day-to-day programming. The language and its implementation can be readily "skinned" with alternative syntax, so I don't worry a whole lot about that.

The current AFL language has very little to say about type issues – neither declarative nor deduced. This makes it weak as a "real" programming language. On the other hand, the absence of type information helps highlight those parts of a programming language that are processing forms – it's in this tutorial role that the current language's strength lies. Much more could be said about the current language and needs to be to make it really useful.

Having trolled the depths of processing forms, it's time to move on to type issues, and their interaction with processing forms. And onto processing forms that require type to make them work: discriminated exception handling, type-identified functions and operators, and rule-based programming. To fit all that in in a clean manner, will require substantial changes to the language – for example, the looping forms will have to be modified to provide for type information without making them too clumsy for use.

My primary motivation in this work, in addition to its explanatory role, is to form a basis for rethinking and illustrating what is needed in our main-stream programming languages in order to make them more text- and markup- (XML) friendly. To too great an extent, the most commonly used languages have designs based on the numeric and system programming needs of the past, and too little on current needs, particularly in the processing of textual data.

1.7 Where AFL Comes From

The design of AFL is inspired by a variety of sources. Coroutines were a major feature of the first object-oriented programming language, Simula[6]. Stackless programming was the chief innovation of the Oregano programming language[2]. Full-function generators are a key feature of the Icon text-processing language[4]. (Icon has coroutines too.)

My own interest and work has largely focused on designing and developing tools, primarily languages, for storing, transmitting and processing text, both free-form and marked-up (e.g. XML). An early language for typesetting was HUGO[3]. A later language for text and markup (primarily SGML and XML) processing is OmniMark[8]. In this work, I and others have had to "fake" coroutines time after time – by storing program state in data state, and having to recreate the program state when resuming a coroutine. This is generally an expensive way of developing software: not only does it add considerable complexity to the initial implementation, but the difficulty of maintaining this form of implementation tends to retard development of such software.

The late arrival of convenient-to-use "pull" parsers for XML[11], following DOM[10] and "push" (a.k.a. SAX[5]) model parsers by almost a decade, is entirely a result of the absence of coroutines in "main-stream" programming languages. In a language that supports coroutines there's no difference between the implementation of the pull and push models. Current pull-model XML parsers are actually only partial implementations of the pull model – they still "push" entity-side and resource requests – a symptom of the difficulty of implementing full pull-model parsing in non-coroutining languages. (I implemented a full pull-model SGML, and later XML parser in the early '90's. It's still used by the OmniMark programming language implementation. I suspect it's the only one of its kind in existence even now. It wasn't an easy job, and was hard to maintain.)

Continuations have been presented as an implementation vehicle by Andrew Appel[1] and others – with the continuations in a lower-level target language than the source language being implemented. They appear in various programming languages including Scheme[7].

Self-defining highly extensible programming languages have been the goal of many language development projects. The most widespread success of this kind has been object-oriented programming: objects allow the user to define their own types. The ongoing appeal of the Lisp family of programming languages (of which Scheme's a good example) is also largely a result of their being easily extensible.

1.8 Getting AFL

The language described by this document is available, together with whatever's happened in the meantime, at <www.wilmott.ca/afl>.

2 The Language: The Familiar

This chapter outlines the parts of AFL that are common in Algol-like programming languages. The familiar – what everyone's seen in other languages – is presented first to help get a feel for the shape of the language.

This chapter primarily describes AFL2, because that's where the most familiar part of the language reside at present.

2.1 The Basics

AFL is an "expression-oriented" programming language: everything other than a name declaration has a result value. (A name declaration is a component of an enclosing value.) Any value can be passed, assigned or returned. Not only does everything have a value, every value has a type, although the AFL2 compiler doesn't know the type of many names.

AFL's syntax is largely a mixture of C and Pascal (actually Algol 60) styles, with a bit of a preference to the latter. Programs are "free form" – line breaks are treated as white space. Comments are treated as white space and can be entered in two forms:

- A "#" (hash or octothorp) that's not a character within a string and anything following it in the same line is a comment.
- A "/*" and everything up to and including the next "*/" is a comment. Nested comments are not (currently) supported.

2.2 Hello World

Many programming language tutorials start out with the language's take on the "Hello World" program: the simplest program that prints out "Hello World". Here's the Hello World program in AFL:

```
"Hello World"
```

The program is one expression: the string "Hello World". Like all expressions, a program has a result value. AFL prints out the result value of a program as a whole if it is a string. Making the last line of a program be the string "OK" or something similar is a simple way of making sure the program has run to completion.

If the result of a program is an integer value, that value is used as the exit code of the program. In any other case the exit code is 1 if the program ends in an exception and 0 if not.

A more traditional form of the Hello World program is the following:

```
print "Hello World"
```

Here what's done with the string is explicated. The result of the "`print`" expression is neither a string nor a number, so "Hello World" is all that's displayed, and the program's exit code is 0. (The "`print`" operator itself returns a value of **nil**.)

There's an even simpler Hello World program that works if you specify `/k=t` on the AFL compiler's command line:

```
Hello World
```

That's "Hello World" without the quotes. This is AFL being used as a templating language.

2.3 Names

All named things are defined in AFL with the **def** declaration:

```
def HelloWorld : "Hello World";
HelloWorld
```

Successive declarations and expressions are separated by semicolons or commas. (User choice, but they can't be mixed in one sequence.)

A defined name using the colon forms is a constant, not a variable. The name "`HelloWorld`" is bound to the value "Hello World". All names are constantly bound to some value, although that value may itself be mutable – a variable. Names are case-sensitive: "`HelloWorld`" and "`helloworld`" are different names.

Some names are "reserved", which means that they are part of the AFL language and are not available for user definition. All names that can be interpreted as number are reserved, as are the keywords of the AFL language. No name is reserved if it is entered using the escaped string form.

2.4 Functions

Functions are defined by following the name in a definition with a list of argument names. What follows the colon is the "body" of the function. Evaluating the body produces the result returned from the function:

```
def f (greeting): greeting ++ " World";
f ("Hello")
```

Functions can be defined recursively:

```
def factorial (n):
  if n == 0 then 1 else n * factorial (n - 1);
```

A function can have zero or more arguments, specified in parentheses.

2.5 Local Scopes

A "local scope" is zero or more definitions, interspersed with expressions, followed by one or more expressions, enclosed in curly brackets:

```
def sqrt (n):
{
  def a = 1;
  def b = n;
  while *a < *b do
  {
    b = (*a + *b) / 2;
    a = n / *b;
  };
  *b;
};
```

Names defined in a local scope are visible throughout the scope (bounded by the curly brackets): there is no "define before use" requirement.

Anything that can be declared can be declared anywhere. For example, functions can be defined inside functions.

Curly brackets are implied around an AFL program as a whole – they can be omitted at the top level.

2.6 Variables

A variable is a name bound to a mutable location in the computer's memory. This relationship is indicated by using the conventional assignment operator in place of the usual colon in a definition:

```
def HelloWorld = "Hello World";
*HelloWorld
```

"=" creates a mutable location and uses the value it prefixes as the initial value of that location. In the example, "Hello World" is the initial value of the "`HelloWorld`" variable.

The prefix "*" (star or asterisk) operator "dereferences" a variable and produces its value. Use of a variable's name without a preceding asterisk always yields the variable itself, suitable for later dereferencing, or assigning to. A variable is a value (a "variable value") and can be passed around like any other value. Assignment is the usual Fortran/C-derived "=":

```
def HelloWorld = "Goodbye cruel world";
HelloWorld = "Hello World";
*HelloWorld
```

If you really want a to declare a variable without initializing it there's an abbreviated form of declaration with no right-hand-side,

which initializes the variable's value to **nil**:

```
def HelloWorld;
HelloWorld = "Hello World";
```

More general definitition of variables and use of the "*" and "=" operators is described in (3.2.7) Variables Revisited.

2.7 Numbers and Arithmetic

You can specify numbers in either decimal or hexadecimal (base 16). A non-negative decimal number must start with a decimal digit followed by zero or more decimal digits and underlines (e.g. 1_000_000). A non-negative hexadecimal number must start with "`0x`", followed by a hexadecimal digit and then zero or more hexadecimal digits and underlines (e.g. 0xFF).

A negative number is specified by prefixing its non-negative value with an underline, as in _1 (i.e. -1). Negative numbers and negating numbers (i.e. prefixing numbers with "-") are kept distinct.

The general arithmetic operations are available: +, -, * and /. Both + and - can be used infix or prefix. The "remainder" operation uses the symbol \. There are two variant division operators: // and \\ are divide and remainder differing from / and \ in that / produces a result rounded towards zero and // produces a result rounded down. \ and \\ give the remainder from the corresponding / and // operators. In the case of \\ it means that its result is always non-negative, a useful operator for doing numeric conversions.

```
def x = 4;
x = 7;         # The result of the program, i.e. its
3 + *x         # exit code, is the number 10.
```

There's an operator for converting values to numbers: `toNumber`:

```
print repr x ++ " = " ++ toNumber x;
```

`toNumber` converts its argument value to a number if it's at all doable: if it's already a number, if it's a .NET enumeration ("enum") value, or if it can be converted to a string with the syntax of a number. If it can't be converted, `toNumber` returns zero. `toNumber` is quite forgiving in its interpretation of string values, accepting C-style and AFl-style, decimal and hexadecimal strings, and allowing commas, spaces or underscores as digit separators.

All the arithmetic operations are available (for variable left-hand arguments) in "do it and assign" forms:

```
+=  -=  *=  /=  \=  //=  \\=
```

2.8 Strings

String literals are specified surrounded by " ("double quotes") with \ (backslash) as the escape characters, as in the C-family languages.

2.8.1 String Operations

There are a few operators on strings and string values:

```
"Hello" ++ " " ++ "World"  # Join strings together.
"*" ** 80                  # Repeat string 80 times.
length "Hello World"   # The number of characters.
"Hello World" take 5   # First 5 characters.
"Hello World" drop 5   # All but the first 5 chars.
repr 5                 # String representation.
char 48          # Unicode char. with given number.
ord "a"          # Unicode number of the character:
                 # arg. must be one character.
toUpper s        # s all in upper-case
toLower s        # s all in lower-case.
tab              # A tab character.
lf               # A line-feed.
cr               # A carriage-return.
```

The arguments of ++ and the first argument of ** are automatically repr'ed, so one can join numbers – with a string result always.

`tab`, `lf` and `cr` are names for the native tab, line-feed and carriage-return characters respectively. These values can be used as-is or interpolated into strings using the \{...} notation, as in these two equivalent expressions:

```
printinline "first" ++ tab ++ "second" ++ cr ++ lf;
printinline "first\{tab}second\{cr}\{lf}";
```

The infix string operations are available (for variable left-hand arguments) in "do it and assign" forms:

```
++=  **=
```

2.8.2 String Literals

Escaping within strings has a mixture of conventional and non-conventional forms:

- \\ and \" represent \ and " respectively. These are the only "built-in" character escapes.
- \ plus zero or more spaces or tabs, plus a line-end is ignored.
- Two or more adjacent strings are joined together and treated as a single string.
- Line breaks are allowed within a string when not in template mode and when not prefixed by \ only if the next following non-empty/non-blank (not even with just a comment in it) line starts with a quote. In that case, the line-end is considered as part of the string, but the following spaces tabs and quote are not.
- \{exprs} within evaluates the expressions, and any definitions within them, and incorporates them within the string if the result is a string or number. If it's a string, it's incorporated as-is, with no requirement for further escaping. If it's a number, it's first converted to its text representation. If it's neither a string nor a number, the result is ignored. How values and definitions are used in string literals is described in more detail in (3.2.2) Strings As Frames.
- \{.} is ignored except where it terminates extended string mode. (Note that "." is not a valid expression, so this sequence is unambiguous in its intent.)
- Any \ not part of one of the above sequences, and any other character other than a \ or " is left "as-is".

2.8.3 Extended and Template Strings

There are three forms of extended and template strings:

- \+ within a string literal says to treat all following characters as string characters up to and not including the four-character sequence \{.} – which terminates this extended string mode.

  Where the \+ is used, if it is immediately followed by zero or more spaces and tabs followed by a line-end, then that white space up to and including the line-end is ignored, so that the extended format string can start at the start of a line – appropriate if it's large and multi-line.
- Any "<" immediately followed by a letter starts an "XML template". An XML template is only terminated by the XML end tag corresponding to that which started the string. The name in the initiating start tag is used to recognize the end tag. Nested uses of the same tag are recognized, as are "empty" tags (start-like tags ending in "/>").
- If template mode is specified on the command line (/k=t), then the whole program is a single extended format string.

Only \{ embedded expressions and \{.} are recognized in template mode. \{.} is ignored unless the template mode started with \+, in which case it returns the string to "normal" mode. In XML templates, /k=t template mode, and in "normal" mode, \{.} is ignored.

2.9 If-Then-Else and Logical Expressions

The **if**-**then**-**else** conditional is more Pascal/Algol-style than C/Java-style, with an explicit **then** keyword and no required parentheses around the condition. **if**-**then**-**else**, like everything **else**, can be used as an expression.

```
def abs (n):
  if n >= 0 then n else -n;
```

if-**then** can be used without an **else** part. **if**-**then**-**else** can be used in an expression. It returns the result of evaluating either the **then** or the **else** part. If there's no **else** part and the **if** part returns false, the **if**-**then** returns **nil**.

Expressions and declarations are separated, not (as in C, Java etc.) terminated by semicolons. There are two cases where this might cause difficulty for those of us used to the C convention:

- A semicolon in front of **else** will separate it from the earlier parts of the **if**-**then** and puts it at the start of a new expression – where it's invalid. For example:

  ```
  if x >= 0 then
    print "x is negative"; # Semicolon invalid.
  else
    print "x is non-negative"; # Semicolon valid
     # if preceding another component expression.
  ```
- A closing curly brace is insufficient to terminate an expression, so a semicolon is need following a curly that ends an expression:

  ```
  if x < 0 then
    {
      x = - x;
    };                # required semicolon
  print "absolute value of x = " ++ x;
  ```

The values **true** and **false** are defined with the standard logical operations:

```
a || b          # a OR b
a && b          # a AND b
! a             # NOT a
```

The infix logical operations are available (for variable left-hand arguments) in "do it and assign" forms:

```
||=  &&=
```

2.10 Comparisons

The six comparison operators are supported:

```
A == B          equal
A != B          not equal
A < B           less than
A > B           greater than
A <= B          less than or equal
A >= B          greater than or equal
```

For == , if the operator's two arguments are of the same type, and are the same value, `true` is returned. Otherwise, `false` is returned. For `!=`, the test made by == is inverted.

For <, >, <= and >=, if the operator's two arguments are of the same type, and are of a comparable type (like string or number), they are compared and `true` or `false` is returned accordingly. If the arguments are of different types or are not comparable, then `false` is returned.

2.11 Loops

There are a variety of looping forms available. Each of the following loops print the numbers from 1 to 10:

```
for 1 to 10 do (i):
  print i;

for 1 by 1 to 10 do (i):
  print i;

returnFrom
  for 1 by 1 do (i):
    {
      print i;
      if i >= 10 then
        return nil
    };

def i = 1;
while *i <= 10 do
  {
    print *i;
    i += 1;
  };

def i = 1;
until *i > 10 do
  {
    print *i;
    i += 1;
  };

def i = 1;
do {
```

```
    print *i;
    i += 1;
  } until *i > 10;

def i = 1;
do {
    print *i;
    i += 1;
  } while *i <= 10;

def i = 1;
loop (exit):
  {
    print *i;
    i += 1;
    if *i > 10 then
      exit '{}
  }
```

The **for** loop specifies the values to be generated between **for** and **do** and the name to which each value is bound each time around in curly brackets followed by a colon after the **do**. Placing the name of the control variable after the **do** keyword is a bit unusual, but there's a reason for it: the scope of the bound name is the expression ("`print n`") it prefixes, not including the expressions between **for** and **do**.

The "while-do", "until-do", "do-until" and "do-while" loops do what they do in other languages.

return exits from a **returnFrom** and provides a value to be returned (**nil** in this case).

The "**loop**" loop loops forever. It specifies an name (in this case "`exit`") that can be used to exit the loop – any name can be used. `exit` is invoked in a special way, like a function, but using '{ ... } (with an opening quote+curly). This distinct notation is used for one-way, non-returning call-like transfers, and is described in more detail in "(4) Continuations". Like the { ... }, non-argument-list form of function call, a value can be expressed within '{ ... } or not. If so, it is returned from the **loop**. If not, **nil** is returned.

Loops can be nested. In the case of the **loop** loop, different levels of **loop** can have different exit names, allowing exiting from outer levels of loop. Alternatively, **returnFrom/return** can be used as an exit mechanism.

2.12 Exception Handling and Finalization

There are times when a program does a non-local exit on you without an explicit initiation. For example, if you divide a number by zero, you're not going to get a result from the division – the program's got to do something else, and that something else is to go somewhere else.

There are two forms of **try** available that give you control over implicit non-local exits: **try** ... **except** and **try** ... **finally**. And ways of explicitly signaling such non-local exits.

2.12.1 Try Except

In AFL you can specify where an implicit non-local exit is caught using the **try** ... **except** construct:

```
try
    # something that might divide by zero or cause
    # an implicit non-local exit in some other way
except (e):
    # to be done for an implicit non-local exit
```

The value passed to the **except** is a .NET Exception object. It's "e" in the example. There are two useful and easily accessible pieces of information available for .NET Exception objects:

- Applying **repr** to an Exception or just printing it returns the text of a useful error message:

  ```
  except (e):
    print e;
  ```

- Applying **typeOf** to an Exception returns the name of the Exception – useful in distinguishing Exception types:

  ```
  except (e):
    if typeOf e != "DivideByZeroException" then
      print e;
  ```

If there's an exception within an **except** part, the **except** outside of the **try** if any, is the one used.

Both the **try** and **except** parts can return values. Like everything else in AFL, **try** ... **except** is an expression. For example:

```
def quotient: try a / b except (e): 0;
```

In this example, the expression following the colon returns the quotient if the divide succeeds, and zero if it doesn't.

2.12.2 Try Finally

Sometimes there's something you want done at the end of a piece of code no matter what happens within that code, there's the **try** ... **finally** construct:

```
try
    # what may terminate with an Exception
    # or return a value
finally
    # something that is done no matter what
```

If the **try** part doesn't terminate with an Exception, its value is the result of the **try**. Whether or not it does, the **finally** part is performed and its value discarded.

2.12.3 Signaling

You can signal an Exception using **signal**:

```
if b == 0 then
  signal "attempt to \{a} / 0";
 a / b
```

signal creates a generic exception with the message text given following the keyword **signal** issues it for capture by outer **try** ... **except** constructs.

2.12.4 Resignalling

If an **except** finds it has an Exception is doesn't want to deal with it can reissue the Exception using **resignal**:

```
except (e):
  if typeOf e != "DivideByZeroException" then
    resignal e;
```

2.12.5 Return

A function returns the value of the expression used as its body. That's often good enough. But sometimes you want to return a value from somewhere inside the function's body:

```
returnFrom
  # Something that returns a value as an expression
  # or uses "return".
```

returnFrom evaluates the following expression, returns its value if there is one, or returns the value supplied by **return** if any. For example:

```
def div (a, b):
  returnFrom
    {
      if b == 0 then
      {
        print "attempt to \{a} / 0";
        return 0;
      }
      a / b;
    };
```

Unlike other languages, AFL decouples functions and returning, so a return can be done out of nested program logic. A return can also exit multiple functions, if the **return** occurs nested more deeply in function calls than its corresponding **returnFrom**.

2.12.6 Lots of Dots

"**...**" (that is the name consisting of three dots) is a no-argument operator that signals an exception. "**...**" is useful in a number of ways:

- It serves as an active run-time "not reached" marker – if you ever get there you get told about it.
- It can serve as a "stub" – you can use it as the definition of a function that you haven't yet got around to implementing:

  ```
  def TranslateFromElvishToKlingon (text): ...;
  ```

2.13 Records

A record is a set of names bound to values. A record is defined by a set of definitions enclosed in curly brackets:

```
def r :
  {
    def field1 : "Hello";
    def field2 : "World";
  };
print r.field1 ++ " " ++ r.field2;
```

Accessing record fields is done by suffixing a record-valued expression with a period and name: the value to which the name is bound is the value returned.

A record value is always a reference: passing and assigning such a value doesn't copy the record.

If a variable contains a record reference value, the record's fields need a dereferencing first. Field selection binds more tightly than dereferencing (actually field selection binds more tightly than almost anything else) so parentheses are required to make the dereferencing happen first:

```
def rr =
  {
    def field1 : "Hello";
    def field2 : "World";
  };
print (*rr).field1 ++ " " ++ (*rr).field2;
```

Records can have creators defined for them by having a record be the result of a function. For example, the following function returns a record with "`re`" and "`im`" fields:

```
def complex (r, i):
 {
   def re : r;
   def im : i;
 };
```

Records can have "destructor/terminator" logic defined for them using the **atEnd** keyword. **atEnd** is placed following the definitions of the record and is itself followed by the expressions providing the record termination logic:

```
def complex (r, i):
 {
   def re : r;
   def im : i;
   atEnd
     print "complex (\{re}, \{im}) is about to "
           "disappear";
 };
```

The empty record can be specified as {} (i.e. curlys with no definitions). Alternatively, the keyword **nil** returns an empty record value (always the same one, which is useful for comparison).

2.14 Dynamic Array Types

One can construct lists, trees and such using records. For other commonly-used aggregate data types, AFL2 supports two further mutable aggregate types: `arraylist` and `hashtable`.

2.14.1 ArrayList

An `arraylist` is a one-dimensional, zero-indexed, variable-sized array – a vector. A new arraylist is created by the "`arraylist`" function. The resulting arraylist has a number of operations defined on it:

```
def a : arraylist (0);  # Create an arraylist
                        # with a zero origin index.
a [] = 3;          # Add an element to the end.
print *a [0];      # Get the item at the indicated
                   # index (needs a dereference).
a [1] = 7;         # Insert a new item at the
                   #  given index (doesn't replace).
print *a [];       # Get the last-most item.
a.remove (0);      # Remove item at the index.
a [0].remove ();   # Thee same thing.
a [].remove ();    # Remove the last-most item.
```

```
print a.count (); # The number of items.
print a.lowbound () # Index of the first item.
print a.highbound () # Index of the last item.
```

A useful alternative form of adding items to an `arraylist` uses the alternative indexing syntax:

```
arraylist (0) '[now]
```

The '[...] adds the enclosed value to the given `arraylist`, and returns the original `arraylist`. This allows more items to be added, and for the `arraylist` to be used in other ways. The '[...] operator can also be used stand-alone. The following expression is equivalent to the last one:

```
'[now]
```

Multiple uses of '[...] can be chained, adding each value to the created arraylist. For example, this definition creates a 3 item arraylist and passes it as the function call's second argument:

```
mailTo (message, '["george"] '["fred"] '["harry"])
```

Although the `arraylist` function is used in these examples to produce the original value, any `arraylist` value expression can be used.

In addition to the above operations, each arraylist has a generator yielding the arraylist's items:

```
for a.generate do (v):
  print v;
```

Each of the values of that `arraylist` are generated in succession, and bound to the name in the "do (...):". See (3.7) generators for more on using generators, especially the useful "each" that allows:

```
for each a do (v):
  print v;
```

2.14.2 Arraylist Origins

The `arraylist` creator function takes a numeric argument. This value is used as the created arraylist's "origin": the index number of its first value, if any.

For the last three decades, zero has been the conventional index of the first item of an array. There are uses for other first-item indexes. For example, many sequences of things are more naturally counted from 1 rather than zero. Requiring the origin to be specified when an arraylist is created, even when it's zero, seems a small price to pay for the ability to choose a different origin when appropriate.

The ".lowbound" arraylist method always returns the arraylist's origin index value.

2.14.3 Hashtable

A `hashtable` is a one-dimensional, key-indexed, variable-sized array. A new hashtable is created by the "`hashtable`" function. The resulting hashtable has a number of operations defined on it:

```
def h : hashtable ();
h ["alpha"] = 3;             # Add or replace item.
print *h ["alpha"];          # Get value with key.
if h.contains ("alpha") then print "Has alpha";
  # True if the hashtable has a value with the key.
h.remove ("alpha");          # Remove item at key.
h ["alpha"].remove ();       # The same thing.
print h.count ();            # Number of items.
```

A useful alternative form of adding items to an `hashtable` uses the alternative indexing syntax:

```
hashtable () '["bk", "urn:sample"]
```

The '[...] operator adds the enclosed key/value pair to a `hashtable`, and returns the original `hashtable`, allowing more key/value pairs to be added, or allowing it to be used otherwise. The '[...] operator can also be used stand-alone. The following expression is equivalent to the last one:

```
'["bk", "urn:sample"]
```

Multiple uses of '[...] can be chained, adding each key/value pair to the created hashtable. For example, this definition creates "`hextable`" with 16 key/value pairs:

```
def hextable :
       '["0",  0] '["1",  1] '["2",  2] '["3",  3]
       '["4",  4] '["5",  5] '["6",  6] '["7",  7]
       '["8",  8] '["9",  9] '["A", 10] '["B", 11]
       '["C", 12] '["D", 13] '["E", 14] '["F", 15];
```

In addition to the above operations, each hashtable has a generator yielding the hashtable's key/value pairs:

```
for h.generate do (k, v):
  print "h [" ++ k ++ "] = " ++ v;
```

Each of the key/value pairs of that `hashtable` are generated in succession, with the key and the value bound to the two names in the "do (...):". See (3.7) generators for more on using generators, especially the useful "each" that allows:

```
for each h do (k, v):
  print "h [" ++ k ++ "] = " ++ v;
```

2.15 Parentheses

Any value can be placed in parentheses. This is usually done where operator precedence would otherwise get the evaluation order wrong, as in:

```
c = (a + b) / 2;
```

It can also be used to group a sequence of expressions. The result of the sequence is the last expression:

```
n = (instanceCount += 1; instanceCount);
```

Because everything (other than a declaration) is an expression in AFL, everything can be parenthesized. For example, the following is valid:

```
def i = 0;
def f (n): n * 2;
(i) = (f) (n); # Variable and function name
               # parenthesized.
```

2.16 Parentheses, Braces and Brackets

AFL makes heavy use of parentheses (...), braces {...} and brackets [...] to express all the syntactic forms it supports. In fact, three aren't enough, so AFL also supports a variant of each of these with a prefixing apostrophe on the opener: '(...), '{...} and '[...]. There are three primary contexts in which each of these forms can appear, with distinct meanings in each case, making for 18 potential combinations:

form	**in a definition header**	**as an expression**	**as a suffix**
(...)	function argument list	parenthesization	list-form function call
{...}	function argument	local scope record or frame	non-list-form function call
[...]	operator property indexer	anonymous indexing	qualified indexing
'(...)	not (yet) used	tuple expression	not (yet) used
'{...}	continuation	template value	continuation call
'[...]	alternative indexer	anonymous alternative indexing	alternative indexing

Where:

- "in a definition header" means between **def** and the following ":", as in:

```
def f (n):
def g {a} {b} {c}:
def [150 a] + [151 b]:
def [x,y]:
def now []:
```

- "as an expression" means where the form is all of a value expression, as in:

```
x = (*y + 3) * 7;
(i) = *(j);
def complex (r, i) : {def re : r; def im : i};
print "fourth item = " ++ [3];
def list : '("a", "b", "c", "d");
```

- "as a suffix" means where an argument, argument list or indexer typically occurs, as in:

```
writeline ("Hello World");
writeline {'("Hello World)};
print "a [3] = " ++ a [3];
hashtable () '["bk", "urn:sample"]
```

This is a bit much for a "real" programming language, but helps serve AFL's intended use as a language for experimentation.

2.17 Semicolons and Commas

Semicolons and commas separate adjacent expressions. They are required where such a boundary isn't indicated by some other keyword or parenthesization. Their primary use is in curly-brace and parenthesized grouping where there are one or more expression in a row, or where there's an expression following a declaration that itself ends in an expression.

There are number of other places where a semicolon or comma is allowed but not required:

- at the beginning or ending of a ( ), { }, [ ], '( ) or '{ } group,
- immediately prior to the **def**, **atEnd**, **include** or **externalModule** keyword,
- immediately after the **atEnd** keyword,
- immediately after the string following the **include** or **externaModule** keyword, and
- immediately following another semicolon or comma.

So all the following are allowed:

```
(;;;)
{;x;}
{x; def y : 1; y;}
{x def y : 1; y}
{;;x;; def y : 1;; y;;}
```

Colons and semicolons may be used interchangeably. But you can't use both in the same sequence. Both the following are allowed:

```
f (x,y,z)
f (x;y;z)
```

but the following isn't:

```
f (x,y;z)      # invalid
```

As was noted earlier, semicolons and commas are separators, not terminators, so the following is invalid:

```
if x > 0 then
  y = x;         # Invalid: it puts the "else" in
                 # a separate expression.
else
  y = -x;        # Valid if not followed by another
                 # "else" or the like.
```

2.18 Include

Anywhere in an AFL program, you can put an **include** command:

```
include "patterns.afl";
```

An **include** incorporates the text of the specified file in the program where it occurs.

An **include** separates expressions. The following is invalid:

```
def f : include "fndef.afl";     # invalid
```

The effect of this definition can be achieved by parenthesizing or curly-bracing the include:

```
def fndef : {include "fndef.afl"};  # valid
```

The latter definition includes the named file (assumed to contain nothing but definitions), but makes all its definitions parts of the "`fndef`" record, meaning they need a "`fndef.`" prefix for access outside of that record. The explicit **self** (described latter) ensures that it is the record that is bound to the name "`fndef`" and not some other value produced within the included file.

There are two options that can be specified for an **include**d file:

- If the file specification is prefixed with a "?" then the **include** is "conditional": if the file has already been **include**d, whether conditionally or not, then it is not **include**d. Conditional includes help ensure that what an **include** defines is not multiply defined. For example:

  ```
  include "?patterns.afl";
  ```

- If the file specification is prefixed with a "+" then the **include** is treated as a (2.8.3) template string:

  ```
  printinline (include "+mytemplate.txt");
  ```

The two options can be combined ("?" must come first), but care has to be made that doing so makes sense:

```
printinline (include "?+mytemplate.txt");
```

3 The Language: The Not-So-Familiar

The non-familiar is what's not found in commonly-used programming languages. It's where AFL starts to get strange but also where underlying ideas start to get explained.

3.1 More About Names

A name in AFL0 can be any of the following:

- Any sequence of one or more letters, digits, "_" or "$".
- Any sequence of one or more of the symbolic characters "~", "`", "!", "@", "$", "%", "^", "&", "*", "-", "+", "=", "|", ":", "<", ">", "?", "/" and "\".

  The character "$" can appear in either the letter/digit form or the symbolic form of a name. You can't otherwise mix letters/digits with symbols unless you quote the name.
- Any sequence of one or more periods (".").
- A string prefixed by a single quote. e.g. '"Hello World". In this example, the space is part of the name. Prefixing doesn't produce different names. The following two names are the same:

  ```
  HelloWorld
  '"HelloWorld"
  ```

- A name that would be otherwise interpreted as a number prefixed by a single quote. e.g. '1 is a name not a number.

  A name that would be otherwise interpreted as a number need not be prefixed by a single quote if it immediately follows a period – i.e. if it is explicitly a field name. In field definitions, the single quote form is required.

All the following are valid names:

```
HelloWorld
_
__0
1st
+
++++++
...
'"1"
'""
'1
$<$
```

Clearly, there's a great deal of opportunity for silliness. On the other hand, there's opportunity for legitimate experimentation.

Reserved names are anything that is a valid number, plus the following names:

```
, . atEnd catch ccallWithEH contArg
def exceptionHandler externalModule
include nil operator throw self
```

There are two names that require care in their use:

- The sequence "/*" can be interpreted as either the start of a comment or as a name or part of a name:
  - If "/*" is found where white space is allowed or required it's the start of a comment.
  - If "/*" is found within a name, it's part of that name.
  - Name quoting forces interpretation as a name:

    ```
    '/*     '"/*"
    ```

- The sequence "<" followed immediately by a letter can be interpreted as either the start of an XML string or as two separate names, one ending in "<" and the next starting in the letter:
  - If "<" is found within a name, as in "<<a", it's part of that name.
  - If "<" followed by a letter is found elsewhere, it's an XML string.
  - Anything other than a letter following a "<" makes it a name. Putting spaces around infix operator uses is a good convention.
  - Name quoting forces interpretation as a name:

    ```
    '<a      '"<"a
    ```

  In both these cases, there are two names in a row: "<" and "a".

3.2 Frames

Local scopes and records look very much the same in the way they are defined and what can be defined within them. They are surrounded by curly braces, the names of things within them are specified using **def**, they can have an **atEnd** part, and anything can be defined within them. This similarity is because they are the same thing. They are both "frames".

A frame is one of the two core kinds of type in AFL. (The other is the (4) continuation, described later.) A frame is a set of named properties, returning a result, with optional associated termination logic. The primary difference between a local scope and a record is what value is returned:

- A local scope follows its definitions with one or more expressions, the last of which is the result value of the local scope.
- A record has no stated result value. Its result value is the frame itself, with its named properties accessible using that value.

3.2.1 Defining and Using Frames

Frames may contain interspersed expressions and declarations: expressions can precede declarations. For an expression's value to be

the result of a frame, the expression must follow all other declarations and expressions in the frame. If the last thing in a frame (other than an **atEnd**) is a declaration, the frame's result is the frame itself.

A frame can explicate its result value using the keyword **self**:

```
def r :
  {
    def field1 : 1;
    def field2 : "Hello World";
    self
  };
```

self can be used anywhere within a frame. Care, however, needs to be taken in its use, because a lot of AFL is implemented by rewriting higher-level functionality in terms of lower, including frames. This can make **self** refer to something not expected. For example, the following will not do the expected thing, because **self** within the function refers to the local frame used to hold function instantiation properties:

```
def r :
  {
    def field1 : 1;
    def field2 : "Hello World";
    def getSelf (): self;  # does the unexpected
    self                   # does the expected
  };
```

Frames and their names in AFL are tied to any code appearing within them. For example, the following frame (within the "`newCounter`" function) returns the function "`c`", so that "`newCounter`" returns that function value. Each invocation of "`newCounter`" creates a new frame with its "`n`" initialized to 0, and it returns a new instance of "`c`". As a consequence, "`anotherNumber1`" and "`anotherNumber2`" are each bound to a different instantiation of the function, and each returns its own sequence of numbers:

```
def newCounter :
  {
    def n = 0;
    def c ():
      {
        n += 1;
        n
      };
    c # the function c is bound to name newCounter
  };
def anotherNumber1 : newCounter ();
def anotherNumber2 : newCounter ();
```

A simple way of remembering what's going on is that if a piece of code to which you can get can see a name, that name is still "alive".

At present there's no concept of "privacy" for frames. However, consider the following definition:

```
def r :
  {
    def a : 1;
    {
      def b (): a;
    }
  };
```

The inner frame is returned by the outer frame. So it's the inner frame that's bound to the name "`r`" – the frame with the property "`b`". The name "`a`" is inaccessible outside of these two frames: it's effectively private to the frame/record bound to "`r`".

3.2.2 Strings As Frames

As noted in (2.8.2) String Literals, within a string literal or template, \{ ... } can be used to embed a string or numeric expression within the string value. More particularly, the following happens:

- For each string literal, including XML templates and a program in /k=t mode, a frame is created.
- Within a \{ ... } embedded in a string literal or template there can be one or more expressions and definitions. The expressions and definitions are part of the string literal's frame.
- For each \{ ... } that returns a value (i.e. does not end in a definition), the value is examined and incorporated in the string value as follows:
  - If the value is a string value, then that string value is incorporated in the overall string as-is.
  - If the value is a numeric value, then that number is converted to its text representation and that text is incorporated in the overall string.
  - If the value is other than a string or number, then the value is ignored, and nothing is added to the overall string.
- The value returned by the frame is the sum of all the literal characters and \{ ... } results within the string literal or template.

There is no provision for an **atEnd** part for a string literal or template.

3.2.3 Frames As Strings

Every frame can function as a template. Within a frame, you can add values to its overall string value, and access the current value of its string value:

- A value enclosed in '{ ... } (quote curly brace) is added to the frame's string value. The enclosed value is a sequence of expressions, not a frame.
- An empty '{ } returns the current frame's string value.

So, for example, the following two expressions are equivalent in their effect:

```
"a\{def x : "c"}b\{x}d"
{'{"a"}; def x : "c"; '{"b"}; '{x}; '{"d"}; '{}}
```

3.2.4 Self

self returns the current frame.

self is a transient value and needs care in its use. Defining and using functions, operators and **catch** can affect its value.

A good general way to avoid these difficulties, where the value of **self** is needed, is to bind the value of **self** to a named property of a frame at the start of a frame prior to any functionality that might cause difficulty, as in:

```
{
  def this : self;
  def f () :
    g (this) # "this" is the outer frame
}
```

3.2.5 AtEnd

At the end of a frame, whether it explicitly specifies a result value or not, one can place the keyword **atEnd**, followed by one or more expressions. These expressions are performed after the frame is discarded. It can be used to ensure some set of actions will be performed no matter how the frame comes to completion:

```
{
  # do something that may exit with an exception
  atEnd
    print "do something that's needed no matter "
          "how you exit";
};
```

The expressions following the **atEnd** are performed no matter what.

3.2.6 Calling Frames

You can use a frame as if it were a function if it has a method named "0callable", as in:

```
def nextNumber :
  {
    def n = 0;
    def 0callable ():
      {
        n += 1;
        n
      }
  }
```

When you go something like:

```
 print nextNumber ();
```

the next value of "n" will be generated and returned from "nextNumber". Callability is closely akin to "closure" – the above frame/function could have been written:

```
def nextNumber :
  {
    def n = 0;
    ():
      {
        n += 1;
        n
      }
  }
```

with the same effect, excepting only that, outside of the "nextNumber" frame, you couldn't go:

```
 print *nextNumber.n;
```

i.e. you couldn't access the frame's properties. This is desirable a lot of the time. Callability is for when it's not.

The value of the "0callable" property of a frame can be another frame, in which case it is in turn examined until a "0callable" property with a function value is found (and invoked). It is an error to attempt to "call" a frame without it having a "0callable" property, or whose "0callable" property is neither a function nor a frame.

The "0callable" property is a Python thing. It's Python (not quite) equivalent is used to aid in defining overloaded properties of what are otherwise function values. In the absence of a declared types and overloaded operators, callability isn't as useful in AFL, but, like other features, it allows for experimentation.

3.2.7 Variables Revisited

(2.6) Variables describes basic variables in AFL, created using the "=" form of name definition, as in:

```
def x = 3;
```

Variables don't actually need to be tied to names – they are values like anything else. The "ref" operator creates a variable value. The following is equivalent to the above definition:

```
def x : ref 3;
```

What "ref" returns (i.e. what a variable value is) is a frame with two properites:

- a function named "get", which takes a list of zero arguments, and returns the current value of the variable, and
- a function named "set", which takes a list of one argument, and sets the current value of the variable to the passed argument value.

So the following two value getters are equivalent:

```
*x
x.get ()
```

as are the following two setters:

```
x = 7;
x.set (7);
```

A final important observation about variables is that any record with appropriately defined "get" and "set" properties can be used as a variable, and can use the "*" and "=" operators. This is used in the implementation of arraylist and hashtable.

3.3 Tuples

A tuple is a fixed-sized ordered sequence of values. AFL provides a useful notation for specifying a tuple value:

```
def t : '(1, 2, 3);
```

A tuple is immutable: it's items can't be changed once the tuple is created.

The items of a tuple are accessed using numeric field names:

```
print t . 1;
print t . 2;
print t . 3;
```

A tuple, like any named set of values in AFL, is a frame. The above definition of "t" is equivalent to:

```
def t : {def '1 : 1, def '2 : 2, def '3 : 3; self};
```

The '(that opens a tuple is a single symbol: no space is allowed within it.

3.4 Functions

Functions need not be named. You can specify a function value with an argument specification, colon and function body:

```
(x,y,z): x + y + z
```

The following two definitions are equivalent:

```
def f (x,y,z): x + y + z;
def f : (x,y,z): x + y + z;
```

The list form of defining and calling functions uses parentheses to surround the argument names in a function definition and the argument values in a call. AFL also supports an alternative, more primitive form of function definition and call that uses curly braces:

```
def abs {x}: if x < 0 then -x else x;
print abs {7}:
```

The difference between the two forms is that the curly brace form passes a single argument value in place of a list of arguments – not a list of one argument. What a parenthesized call does is pass its list of argument values as a tuple, so the following two calls are the same:

```
f (x,y,z)
f {'(x,y,z)}
```

On the definition side, the passed tuple is broken up into its named components. These two definitions do the same thing:

```
def f (x,y,z): x + y + z;
def f {0args}:
  {
    def x : 0args . 1;
    def y : 0args . 2;
    def z : 0args . 3;
    x + y + z
  };
```

There are a number of further shorthand notations for use in defining functions:

- In defining or calling a function with curly braces, you can omit the argument name or argument value. In the definition this makes the argument anonymous, so you can't get at it. In the call it causes the default value of **nil** to be passed:

  ```
  def f {}: print "f called";
  def g : {}: print "g called";
  f {};
  g {};
  ```

- In the anonymous or **def** function form you can have multiple argument lists. Each list defines a function. The following three definitions are equivalent:

  ```
  def f (x) (y,z): x + y + z;
  def f : (x) (y,z): x + y + z;
  def f : (x): (y,z): x + y + z;
  ```

  In each of these cases, calling "f" with one argument returns a function that takes two arguments. A call can be like the following:

  ```
  print f (1) (2, 3);
  ```

3.5 Operators

None of the operators in AFL are actually part of the core language. They are defined using AFL's operator definition syntax, and are implemented in AFL, in AFL's run time library, or using functionality provided by .NET's libraries.

There are three kind of operator definitions: with arguments, without arguments, and indexing.

There are no overloaded operators – that requires a static type system (which is in the works).

3.5.1 Operators With Arguments

Operators are defined with an extended form of the **def** declaration:

```
def - [151 a] : 0 - a;
def [160 a] // [161 b] :
          if a \ b < 0 then a / b - 1 else a / b;
def [190 n] ! : if n == 0 then 1 else n * (n - 1)!;
```

An operator definition consists of alternating operator part names and argument specifications. Prefix, infix and postfix operators can be defined, as can multi-part operators.

An argument specification is placed in square brackets, and consists of a precedence number followed by an argument name. A higher precedence number causes the operator to bind more tightly. For example:

```
- 1 // 2     # Is evaluated as
- (1 // 2)   # and not as
(- 1) // 2   # because // binds more tightly
             # (160/161) than does - (151).
```

The relative values of right and left arguments determines whether an operator is left- or right-associative.

An operator defined within a frame can only be used within that frame and within nested frames. You can't use an operator name as a field name. The following is invalid:

```
def r :
  {
    def a = 1;
    def ++ [180 b]: a + b;
  };
print r.++ 7;  # invalid
```

Operators are about syntax. Their names are place-holders in the syntax they define, not the names of things.

The functions that operator definitions are rewritten to by the AFL compiler do have names, that can be used in the qualified form. These are described later: see (3.5.6) Operator Names.

3.5.2 Delayed-Evaluation Operator Arguments

A third component can be used in operator argument specifications – an empty pair of parentheses following the argument name, as in:

```
def if [90 a] then [90 b ()] else [90 c ()]:
    a (b, c) ();
```

If () is specified in this way then an argument value is not evaluated at the point of call but rather is wrapped as the body of a function with a zero-length argument list. The function argument value is passed to the invoked operator, wherein the value can be found by calling the function (the last () in the "if" example).

3.5.3 Grouped Operator Arguments

In place of a precedence number in an operator argument specification, there can be an empty group – any of (), {}, [], '(), '{} or '[] – indicating that the argument must be wrapped with the given delimiters, as in:

```
def if [() a] [90 b ()] else [90 c ()]:
    a (b, c) ();
```

When a grouped argument is specified and it's not the last argument, then the following argument must be specified immediately, without an intervening name.

The sample "if" definition places the condition in a group, meaning there's no "then" keyword. This allows use of the C style of composite statement syntax:

```
if (a < 0)
  print "negative"
else
  print "non-negative";
```

There can be more than one grouped argument in a row. We could have defined "if" as:

```
def if [() a] [{} b ()] [{} c ()]: a (b, c) ();
```

in which case the "else" keyword would go and the then and else parts would need to have {...} grouping, as in:

```
if (a < 0)
  {print "negative}
  {print "non-negative"};
```

In this latter example, the { ... } around the then and else parts are there to wrap the arguments. They don't make the arguments into frames. A second set of curlys would be required to do that:

```
if (a < 0)
  {{print "negative}}
  {{print "non-negative"}};
```

which is why this is probably not a good syntax in this case.

The particular kind of grouping specified in an argument is used when identifying an operator. For example, the following two definitions are distinct and valid:

```
def a [() x]: print "a (x) = " ++ x;
def a [{} x]: print "a {x} = " ++ x;
a ("hello");
a {"there"};
```

and the program's output is:

```
a (x) = hello
a {x} = there
```

making it clear that "a (...)" and "a {...}" are different operators. (I haven't come up with a case where this distinction is useful yet, but hey, experimentation.)

3.5.4 No-Argument Operators

There's one other form of operator definition, for operators with no arguments. (You can define operators with one, two etc. arguments, so why not zero?)

```
def a = 0;
def 2xa []: 2 * *a;
```

The operator "2xa" is invoked as if it were a name, not an operator or function. Each time it is used, its definition is invoked:

```
print 2xa;
```

The syntax for the no-argument operator definition is a bit inconsistent. The square brackets are used for consistency with other operator definitions, but there's nothing to put in them. All they do is distinguish this form of definition from other forms. (No-argument operators are useful for experimenting with a number of ideas. The current definition syntax and naming logic isn't entirely satisfactory, and needs a fix.)

3.5.5 Indexing

There's another useful kind of function/operator that can be defined in AFL, indexing:

```
def pair (a, b):
  {
    def left : a;
    def right : b;
    def [a] :
        if a > 0 then right [a - 1] else left;
  };
def myList:
    pair (100, pair (200, pair (300, nil)));
def getThird (a): a [2];
print getThird (myList);
```

The definition of [...] in the record created by "pair" returns the "nth" item in a chained list built using pairs. It's invoked using a subscripting-like syntax.

The definition of the indexer uses square brackets as the indexer "name". This is not the operator argument syntax because what's in the square brackets doesn't start with a precedence number. There can be more than one indexer defined in a scope, so long as each has a different number of arguments defined for it:

```
def [] : if right == nil then left else right [];
         # Last item right-branch-wise.
def [a, b]: if a > 0 then right [a - 1, b] else
           if b > 0 then left [0, b - 1] else left;
```

```
# 2 dimensional indexing of tree.
```

Indexers work somewhat differently from other operators in that they are usually accessed outside of the scope in which they are defined. When used within the scope in which they are defined, indexers can be used without a prefixing qualification:

```
[2]          # get 2nd (or 3rd if zero-based) item
self [2]     # equivalent
```

There's a second form of indexer, using '[and] as brackets – the opener is a two-character symbol, consisting of single quote and open square bracket. This second form is defined and used in just the same manner as the first.

3.5.6 Operator Names

As noted in the description of operators with arguments, operator names are not names of things but are syntactic place-holders. However, all operators and indexers do have names, which can be used in the usual way:

- A function name is constructed for each operator with arguments out of its syntactic place holder names. These names are joined into a single name as follows:
  - If there is more than one syntactic name, the partial names are joined with a "$" between each pair.
  - If there is an argument prior to the first name (i.e. an infix or postfix operator) a "$" is placed at the start of the name.
  - If there is an argument following the last name (i.e. a prefix or infix operator) a "$" is placed at the end of the name.

  The resulting name is that of a function that takes a list of the operator's arguments. For example:

  ```
  print $+$ (3, 4);  # prints 7, i.e. 3 + 4
  print -$ (3); # prints _3, i.e. - 3
  print if$then$else$ (a >= 0, a, -a);
              # Prints the absolute value of "a".
  ```

  Where the operator has a () argument, the corresponding function doesn't. So the second and third argument of "if$then$else$" need to be passed as functions and the result of "if$then$else$" invoked if it's not desired to evaluate the failing alternative.
- An operator with no arguments has a function defined with the same name as the operator. The function isn't accessible using an unqualified name. If it's defined within a frame, the frame (e.g. "`self.`") can be used to qualify the name and get the function.
- Indexers have a name consisting of the number of arguments of the indexer (zero or more) followed by "`item`", for the [...] form, or by "`additem`", for the '[...] form. So a 1 item [...] indexer has the name "`1item`". This function name can be used like any other function name, qualified or not. The following two indexings/calls are equivalent:

  ```
  a [b]
  a.1item (b)
  ```

3.5.7 Mixing Operator and Function Definitions

Similarly to function definitions, zero or more arguments or argument list can be specified following an operator prototype:

```
def [150 a] +++ [151 b] (c): a + b + c;
```

The operator +++ returns a function that adds the argument passed to it to those passed to the +++ operator.

Other combinations are supported:

```
def r :
  {
    def [] (): print "OK";
    [] ();        # The first "OK"
    self
  };
def y [180 a] (): print a;
def x [] (): print "Also OK";
r [] ();          # The second "OK"
(y "OK Too") ();  # "OK Too"
x ();             # "Also OK"
```

(Yes, strange and not much else. But consistent. The language design attempts to make things as uniform as possible.)

3.5.8 Separating Operator Syntax and Functionality

A name definition (**def** followed by a name) binds a value or functionality to a name. In contrast, an operator definition binds functionality to a piece of syntax: a prefix, infix or postfix operator, or an indexer.

In a name definition, there's the name and everything after that describes the value bound to the name. But when used to define an operator, there is some overlap between the syntax and the functionality: the function's arguments are declared in between the operator's symbols, within it's syntax, even though they are part of the value. To clarify the syntax vs. functionality issues for operators, AFL has a second form of definition, with the keyword **op** following **def**, as in:

```
def op [150] + [151]: (a, b): a - (0 - b);
```

The difference between **def** and **def op** is that **def op** defines the syntax of the operator followed (after the colon) by what it's bound to – which must be a function value for it to be useful – whereas just **def** actually defines a function. The "+" definition using just **def** is as follows, with the "(a, b):" bit missing because the arguments are declared as part of the operator's arguments' syntax:

```
def op [150 a] + [151 b]: a - (0 - b);
```

What's allowed and not for **def op** are:

- The argument declarations for **def op** are the same as for **def** except that either the argument names must all be specified as asterisk ("*") or omitted.
- For a non-indexing operator, there is no ambiguity in omitting the asterisks, so doing so is allowed.
- For an indexing operator, asterisks must be used. This avoids ambiguity between, for example, the zero and one argument forms:

```
def op []: (): 0;
def op [*]: (a): a;
```

- The right-hand side of a **def op** cannot be defined using the "=" or omitted variable definition form.
- A no-argument operator can be defined without the suffixing "[]", as in:

```
def op arraylist0 : (): arraylist (0);
```

The alternative **def op** form, in addition

- to helping explain what operators really are, and
- making a better syntax for zero-argument operators,

allows for one thing that can't be done with the plain **def** syntax:

- You can bind an operator to an function or operator defined elsewhere, as in:

```
def op [190] ! : factorial;
```

So for every form of operator definable using just **def**, there's a corresponding form using **def op**, although the opposite isn't the case. So in terms of rewrites, the **def op** form can be considered more basic.

3.6 True and False

In AFL, the values true and false are functions, not scalar values. They are defined as follows:

```
def true (a, b) : a;
def false (a, b) : b;
```

`true` is a function that expects two calls and returns the argument value passed it by the first call. `false` returns the second call's argument value. true and false are "selectors": they choose one value or the other.

The most primitive form of conditional is the following, where "a" has a value of true or false:

```
 a (b, c)
```

Depending on whether "a" is `true` or `false`, "b" or "c" will be returned.

It's usually the case in a conditional that there's evaluation required for the chosen **then** or **else** part without evaluating the unchosen alternative. So a better form of the conditional is the following:

```
a ((): b, (): c) ()
```

In other words, postpone evaluating "b" or "c" by making them functions, and call the function returned by the conditional once chosen. This is how the if-then-else operator is defined:

```
def if [90 a] then [90 b ()] else [90 c ()]:
    a (b, c) ();
def if [90 a] then [90 b ()]: a (b, (): nil) ();
```

The logical operations are defined in a similar way:

```
def [120 a] || [121 b ()] : a ((): a, b) ();
def [121 a] && [122 b ()] : a (b, (): a) ();
def ! [122 a] : a (false, true);
```

For "||" and "&&" the first argument is always evaluated but the second one may not be if the first argument is sufficient to determine the result. This is consistent with how these operators are defined in other languages.

3.7 Generators

A generator is a function that retains persistent state. It typically generates a sequence of values, returning each value on successive calls to the function.

3.7.1 Using Generators

The **for** operator takes a generator as its first argument. A generator in AFL is any function or operator that returns a function that takes a function that when called is passed one of the values generated. Hmmm, maybe an example would work better:

```
def generator : 1 to 10;
def display (n): print n;
generator (display);
```

In this example, the name "generator" is bound to an the generator returned by "1 to 10" and "display" is bound to a function that's to be called for each value returned by the generator. The call "generator (display)" invokes the generator, passing it the function to be called for each generated value.

Because of AFL's expression syntax, you can also formulate the above example with no name definitions with the same effect:

```
(1 to 10) ((n): print n);
```

Generators support some of the basic looping control structures in AFL. Here's the definition of **for**:

```
def for [90 G] do [90 B]: G (B);
```

Which means that the above example can be expressed as:

```
for 1 to 10 do (n): print n;
```

The primary role of the operator definition is, like that for if-then-else, to provide a more expressive and readable form of what is a common construct – and it's always good to get rid of parentheses. (Sorry again, Lisp fans.)

3.7.2 Defining Generators

A generator is defined as function that takes a function as its argument:

```
def alphabet (yield):
  for 0 to 25 do (i):
    yield ("abcdefghijklmnopqrstuvwxyz" drop i
                                        take 1);
```

The generator calls the passed function with each value it generates in succession. In this example, "`alphabet`" is the generator and it calls the "`yield`" function each time around. It can be used as follows:

```
for alphabet do (c):
  print c;  # Prints letters one-per-line.
```

Usually a generator is defined with parameterization:

```
def allchars (aString) (yield):
  for 0 to length aString - 1 do (i):
    yield (aString drop i take 1);
for allchars ("abcdefghijklmnopqrstuvwxyz") do (c):
  print c;
```

Note that "allchars" has two argument set specifications: the first parameterizing the generator, and the second used for invoking and running the generator. This two-argument-set form is common for generator definitions.

Generators in AFL are "true" generators, in that they can yield their values not only from the top-level generator function, but from any nested function in which the "yield" function is passed or otherwise made available. This is a major advantage when filtering complexly structured input data.

It's appropriate to define commonly used generators as operators. This is a definition for the by-to, to and by operators:

```
def [120 a] by [121 s] to [120 z] (yield):
  catch exit:
    {
      def loopup '{n} : (
        if n > z then exit '{};
        yield (n);
        loopup '{n + s}
      );
      def loopdown '{n} : (
        if n < z then exit '{};
        yield (n);
        loopdown '{n + s}
      );
      (if s >= 0 then loopup else loopdown) '{a};
    };
def [120 a] to [120 z]: a by 1 to z;
def [120 a] by [121 s] (yield):
  {
    def loop '{n} : (
      yield (n);
      loop '{n + s}
    );
    loop '{a};
  };
```

(4) Continuations are used to define by-to and by because they're primitive forms for use by other definitions – and one has to start somewhere. Most user definitions of generators will use lower-level generators in their definitions rather than continuations – as did the "alphabet" and "allchars" examples above. (I've been trying to postpone talking about continuations, but they just snuck in.)

3.7.3 Generator Operators

There are a number of operations defined on generators:

`generator1 :++: generator2`

> where generator1 and generator2 are themselves generators. Generates all that generator1 generates and then all that generator2 generates. The number of values generated by `:++:` is the number of values generated by generator1 plus the number of values generated by generator2. `:++:` is the "sum" generator.

`generator1 inner generator2`

> where generator1 and generator2 are themselves generators. Successively generates two-item tuples (pairs) of the values generated by generator1 and generator2 in parallel, terminating when either generator1 or generator2 ceases. The number of values generated by `inner` is the lesser of the number of items generated by generator1 and the number of items generated by generator2. `inner` is the "inner product" generator.

`generator1 outer generator2`

> where generator1 and generator2 are themselves generators. Successively generates two-item tuples (pairs), by pairing each value generated by generator1 with every value generated by generator2. The number of values generated by `outer` is the number of items generated by generator1 times the number of items generated by generator2. generated. Unlike `inner`, which invokes each generator once, `outer` also invokes the generator1 once, but invokes the generator2 once per value of the first generator1. `outer` is the "outer product" generator.

`generator :-: count`

> where generator is a generator and count is a number. Generates what's generated by generator, but skips over the first count values generated by generator, generating only the values following the first count.

`generator :+: count`

> where generator is a generator and count is a number. Generates the first count values generated by generator, not generating any values following the first count.

`each value`

> where value is a frame with a "generator" property, that is a function with a one-item argument list. Returns that "generator" property. "each" supports things like "for each myArraylist do (v): print v".

`value forever`

> value is yielded over and over again. "forever" is most use in conjunction with explicit loop exits or with other generator operators, as in "x forever :+: 10".

`function map generator`

> where function is a function and generator is a generator. function must accept of the values generated by generator as a valid argument. Successively generates the result of calling function with each of the values of generator. The number of values generated by "map" is the number of values generated by generator.

`initialValue next sequencer to tester`

> where sequencer and tester are functions with a one-item argument list. "next ... to" generates values starting with initialValue, passing them to tester for validation and passing each value to sequencer to produce the next value, as in "0 next (i): i + 1 to (i): i < 10". The values so creted, includ-

ing initialValue, are yielded until tester returns false, at which point "next ... to" terminates. If initialValue fails tester, no values are yielded.

`initialValue next sequencer`

where sequencer is a function with a one-item argument list. "next" generates values starting with initialValue, passing each value to sequencer to produce the next value, as in "0 next (i): i + 1". The values so creted, including initialValue, are yielded in tern. "next" doesn't terminate.

`value once`

value is yielded exactly once. "once" is most use for adding a value to an otherwise generated sequence, as in "each myArraylist :++: nil once".

`ungenerate (generator, exit)`

where generator is a generator and exit is a continuation (e.g. **catch** identifier). `ungenerate` returns a function that on successive calls, returns the values generated by the generator. When there are no further values, exit is invoked.

`generator whilst function`

where generator is a generator and function is a function with a one-argument argument list that returns a **true** or **false** result. "`whilst`" takes each value generated by generator, it to function and generates that value if function returns **true**. "`whilst`" terminates if function returns **false**.

As an example of using generators to define other generators, AFL defines the `:++:` operator as follows:

```
def [135 a] :++: [136 b] (yield):
  (
    a (yield);
    b (yield);
  );
```

An example:

```
print "Non-multiples of 10:";
for 1 to 9 :++: 11 to 19 :++: 21 to 29 do (n):
  print n;
```

3.7.4 True vs. Simple Generators

Languages like Python and C# support generators. Generators in those languages are called "simple generators".

A simple generator is a function, as in AFL, that can successively "yield" values, and which terminates when it returns. AFL is different in that yielding is not limited to the top-level generator function, but can be done from anywhere – most importantly from within functions called by the top-level generator function itself. AFL's generators are "true generators". (AFL is not unique in having true generators, it's just that the languages used by most programmers don't have them.)

Simple generators work well for linearly structured input – where there's one thing after another. But something more is needed when the input has a non-linear structure – a large variety of data formats have nested structures, for example. It's not that simple generators are dead-in-the-water with respect to complex structure. Simple generators can invoke nested generators and re-yield their results, and this can deal with a lot of nested structures. It's just that true generators allow for a wider range of possibilities.

3.8 Coroutines

Coroutines are the single most important general programming language functionality absent from major programming languages.

A coroutine is an independent but synchronous thread of execution. What's traditionally called a "thread" is in every way like a coroutine, excepting only that a traditional thread is asynchronous. In practice, this small difference makes a big difference in how traditional threads and coroutines are best used, and in their performance – switching between coroutines, appropriately implemented, can take no more time than a simple function call, whereas switching between threads often has a substantial overhead. Because of their potentially low overhead, it is practical to use coroutines for a lot of tasks, especially in the area of text processing.

(Statements about the performance of coroutines are intended to be about coroutines in general, not about the performance of AFL's implementation of coroutines.)

Synchronicity between coroutines is maintained by the simple expedient of one coroutine calling/transferring control to the other. It's the same kind of synchronicity maintained between functions and their callers.

3.8.1 Defining and Using Coroutines

In AFL you create a coroutine by calling the "`makeroutine`" function:

```
def coroutine :
  makecoroutine (coroutineTopLevel, coroutineExit);
```

`makecoroutine` is passed two arguments:

- A function to be called to start the new coroutine running.
- A continuation (e.g. **catch** identifier) to be transferred to if and when the new coroutine exits.

`makecoroutine` returns a function value.

You're always in a coroutine. The "main" thread of a program is a coroutine, but that isn't an important issue unless there are other coroutines around.

The new coroutine's "top level" function is passed two arguments:

- A function that suspends the new coroutine and passes a value to where it was resumed from.
- The first value passed when resuming the new coroutine.

The function returned by `makecoroutine` and the function passed as the first argument of the new coroutine's top-level function are the creator's and new coroutine's "`resume`" functions. They each work just the same:

- When invoked, it suspends the current coroutine and resumes the other coroutine.
- The value passed to the function is returned from the other coroutine's resumer as its result, excepting only that the first

time that the new coroutine is resumed, the value is passed as its top-level function's second argument.

Both resume functions are AFL values, which means that they can be passed around and used as you wish. They are functions, that can be used like any other function. The main consequence of this is that if there are more than two coroutines around, who's resuming a coroutine may not be the one that originally was returned or passed that resume function. But that's OK, a coroutine should just do its job and leave it to the resumer to decide when it should be called.

3.8.2 A Coroutine Example

Here's a simple example of a pair of coroutines passing values back and forth:

```
def cr1 (suspend) :
  {
    print "Starting cr1";
    for 1 to 5 do (n):
      (
        print "cr1 loop passes to cr2, value: \{n}";
        print "cr1 is resumed with value: " ++
              suspend (n);
      );
    print "cr1 exits";
  };

def cr2 (suspend, firstArg) :
  {
    print "Starting cr2 with value: \{firstArg}";
    for 5 by -1 do (n):
      (
        print "cr2 loop passes to cr1, value: \{n}";
        print "cr2 is resumed with value: " ++
              suspend (n);
      );
    # cr2 never exits -- the for goes on forever
  };

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

The last line makes a new coroutine, makes "`cr2`" its top-level function, says that the program should terminate the program if it ever exits, and invokes "`cr1`" with the new coroutine's resumption function.

An important feature of this example is that it illustrates the symmetry of coroutines. What goes on in the loops in `cr1` and `cr2` is the same.

Here's the output:

```
Starting cr1
cr1 loop passes to cr2, value: 1
Starting cr2 with value: 1
cr2 loop passes to cr1, value: 5
cr1 is resumed with value: 5
cr1 loop passes to cr2, value: 2
cr2 is resumed with value: 2
cr2 loop passes to cr1, value: 4
cr1 is resumed with value: 4
cr1 loop passes to cr2, value: 3
cr2 is resumed with value: 3
cr2 loop passes to cr1, value: 3
cr1 is resumed with value: 3
cr1 loop passes to cr2, value: 4
cr2 is resumed with value: 4
cr2 loop passes to cr1, value: 2
cr1 is resumed with value: 2
cr1 loop passes to cr2, value: 5
cr2 is resumed with value: 5
cr2 loop passes to cr1, value: 1
cr1 is resumed with value: 1
cr1 exits
```

3.8.3 currentTextWriter, currentTextReader and currentSource

(5.2.4) currentTextWriter, currentTextReader and (5.3.1) currentSource are the only mutable global values maintained by "include" files supplied with AFL. These values are global: not fields of records, and accessible anywhere. This can cause difficulty when using these values in more than one coroutine.

To make it practical to use these values (implicitly or explicitly), AFL (AFL2 in particular, not the core language) implements these names so that:

- each coroutine has its own copy of each of these values,
- these names always refer to the current coroutine's copy of the named property, and
- each of these names is a reference – that can be safely assigned to in each coroutine.

By default, when a new coroutine is created, these three values are copied from the creating coroutine. After that, they are maintained separately. The (5.2.7) streamFrom and streamInto operators are essentially distinguished from the generic `makecoroutine` by how they establish these three values.

```
currentTextWriter, currentTextReader and currentSource are al
currentTextReader, currentTextWriter and
currentSource in Threads.
```

3.8.4 Benefiting From Coroutines

Coroutines can be tricky to use, and coroutines aren't for everyone. But once a coroutine-base interface has been implemented, the resulting interface can be very easy to use, both for the implementers of packages using the interface, and, more importantly, for users of packages with coroutine-based interfaces. That's because you use coroutines to hide the complexities of implementation within the coroutine-based interface, freeing both the user and package implementer from a lot of interface issues.

A coroutine-based interface works because it doesn't look like a coroutine-based interface. It looks like something else, that the user or package implementer is already familiar with:

- a function or method call,
- using a generator in a "for" loop, or
- piping data processing through a "filter": which is just an expression or function that reads from its default input and writes to its default output.

3.9 Accessing .NET Functionality

Non-AFL objects and their features can be directly accessed from AFL using a variant of function call syntax.

The mechanism described here is something of a place-holder. It requires too much knowledge of the details of .NET arcana for common use, and needs to be replaced with something more user-friendly. However, this mechanism has been a great deal of help in learning about how .NET works, and in providing access to .NET functionality.

3.9.1 Calling .NET Functions

If the function is specified as a literal string value (not a name whose value is a string), then it is interpreted as the specification of a .NET feature. For example, the following call invokes the static method "System.String.Concat", passes it two string values and expects a string value to be returned:

```
"string System.String::Concat(string,string)"
                                        (a, b)
```

AFL examines the string specification and makes sure that the arguments and result are treated properly.

The general syntax of the external function specification consists of:

```
OPTIONS RETURN CLASS::METHOD-NAME (ARGUMENT-SPECS)
```

(Use of "::" instead of "." to separate class and method names is a .NET thing.)

There are two prefixes that can be placed at the start of an external function specification:

- "`new`" specifies that a new object be created and that its creator be invoked. For example, the following creates a new instance of "System.Collections.ArrayList":

  ```
  "new void System.Collections.ArrayList()" ()
  ```

  When creating new object, there are two things to be noted: the result type is always "`void`" because the actual call is to the new object's initializer, and the initializer's method name is ".ctor", which can be omitted (AFL adds it if it's missing).
- "`virt`" specifies that the method is "virtual" rather than "static". In this case, an extra first argument is expected in invoking the method: the class of which it's a member. For example, the following call to the "System.String" class's "`Length`" property expects a string value to be passes as its single argument, even though the specification denotes no arguments:

  ```
  "virt int32 System.String::get_Length()" (a)
  ```

As in this last example, a C# "property" needs to be specified with the appropriate prefix on the name: "`get_`" or "`set_`".

3.9.2 Qualifying .NET Names

In general, .NET classes have to be fully namespace qualified and annotated with their containing module. The "`System`" namespace resides in "`mscorlib`":

```
"virt int32 [mscorlib]System.String::get_Length()"
                                                (a)
```

All modules need to be specified. For example, the following specification describes the "`Plus`" (i.e. "+") operator implementation in AFL's run-time library:

```
def [150 a] + [151 b] :
     "int64 [.module aflruntime.netmodule]"
        "AflRuntime.Afl::Plus(int64,int64)" (a, b);
```

Full qualification of this sort is required for argument types defined in namespaces as well.

Where a .NET name is a value type with a qualified name, the type name and annotation needs to be prefixed by "`valuetype`":

```
"virt object valuetype DictionaryEntry::get_Key()"
                                                (v)
```

3.9.3 Accessing .NET Fields

If there's no argument list specified then the access is to a field, not a method:

```
"class [mscorlib]System.Collections.ArrayList "
    "[.module aflruntime.netmodule]"
    "AflRuntime.Afl::AflArgs" ()
```

This access to "`AflArgs`" is to a static field. The argument list is required in AFL even though it's a field access (in order that AFL recognizes the use of the string as a .NET name). If "`virt`" is specified for a field access, it indicates that the access is to an instance field, in which case the record has to be given in the argument list.

3.9.4 Casting

All argument values of .NET functions are casted or unboxed by AFL as required. Likewise, all result values are casted or boxed to make them AFL-friendly. However, there are occasional cases where further casting is required. AFL support a "cast" form of prototype:

```
cast TOTYPE FROMTYPE
```

The prototype must be invoked with a single argument, which is casted first to FROMTYPE and then to TOTYPE. For example:

```
"cast enum Xml.XmlNodeType int32" (1)
```

3.9.5 Passing and Returning Enums

.NET enumerated values (enums) can be created and read from within AFL. There are a number of facilities for manipulating such values:

- The `repr` operator gives name of the enum value as string.
- The `toNumber` operator gives the numeric value of the enum value.
- The "cast" form of external specification can be used to convert a number to its equivalent enum value:

  ```
  "cast enum Xml.XmlNodeType int32" (1)
  ```

- A string can be converted to the enum value it names using .NET's reflection facilities:

```
def XmlNodeType (n) :
   "object System.Enum::Parse"
                "(class System.Type,string)"
  ("virt class System.Type object::GetType()"
   ("cast enum Xml.XmlNodeType int32" (0)), n);
```

  In this example, the "cast" operation is used to create a value of the appropriate enum type, the "GetType" to get its type, and the Parse to convert the string (n) to the enum value. Any other enum type value can be similarly created simply by replacing the name "Xml.XmlNodeType" with another's enum's type name.

Enums are value types in .NET's type system. However, to make sure AFL does the right thing with such values, the word "enum" must be used in place of "valuetype" in an enum's type specification, as in the above example.

3.9.6 .NET Properties

There are a few special kinds of functions defined in the .NET system, called "properties". These are names defined with "get" and "set" sub-definitions, which are called when the name is used in a source or target context. Properties are indicated by prefixing the .NET names with "get_" or "set_", as in:

```
"virt int32 System.String::get_Length()"
```

There's another kind of property that prefixes a name with "op_": operators. The operator's name (as opposed to its symbol) is prefixed with "op_", as in the definition of the time difference operator definition:

```
def [150 a] -- [151 b]:
  "valuetype System.TimeSpan System.DateTime::"
  "op_Subtraction(valuetype System.DateTime,"
              "valuetype System.DateTime)" (a, b);
```

AFL is very limited in its support for "op_" properties at the moment. In fact, "op_Subtraction" is the only one it recognizes.

3.9.7 External Function Argument Lists

A .NET function's arguments are passed as a tuple when using the parenthesized argument list form. There is also a curly-brace form, which expects a tuple to be passed to it. The following two are equivalent:

```
"string System.String::Concat(string,string)"
                                          (a, b)
"string System.String::Concat(string,string)"
                                        {'(a, b)}
```

This form is useful when it is desired to pass along an argument list as-is:

```
def concat {args}:
  "string System.String::Concat(string,string)"
                                          {args};
```

3.9.8 Null, True and False

.NET's "null" value and AFL's **nil** are not the same thing. Ditto their true and false values. AFL provides access to .NET's null, true and false, primarily so that they can be passed to external .NET functions, and so that if they are returned by a .NET function they can be tested:

`NetNull` the .NET null value.

`NetTrue` the .NET true value.

`NetFalse` the .NET false value.

When an external function returns a "bool" value, it's best to use the AFL-provided operator "`unbool`" on the result of the function. `unbool` converts a .NET bool value into its corresponding AFL value, as in:

```
def [131 a] == [131 b] :
  unbool "bool System.Object::"
                  "Equals(object,object)" (a, b);
```

3.9.9 .NET Name Shorthand

There are two useful abbreviations for this rather lengthy notation. "[mscorlib]" is always optional for the "`System`" namespace. As well, AFL provides an "`externalModule`" declaration:

```
externalModule
    "[.module aflruntime.netmodule]AflRuntime.Afl";
```

This example indicates that any class name starting with "`Afl`" is in the "`AflRuntime`" namespace of the "[.module aflruntime.netmodule]" module.

The interpretation of the string argument of **externalModule** takes the last period-separated part of the given name – in this case "`Afl`" – and uses the whole string as a replacement for that substring when it appears as the first part of a class name.

externalModule can appear anywhere. It is not bound to any scope of an AFL program. **externalModule** separates expressions, so it does have an impact on a program's syntax.

3.9.10 Declaring .NET Assembly Attributes

In addition to the other information provided by external function specification strings and **externalModule** declarations, AFL allows .NET assembly attributes to be specified. This is required when a library's identification is based on version information as well as name. The syntax for doing this is a variation of the **externalModule** declaration, wherein the string consists of a .NET namespace name followed by attributes in curly brackets, as in:

```
externalModule "System.Xml{.publickeytoken="
               "(B77A5C561934E089) .ver 1:0:3300:0}"
```

(This is definitely over the edge when it comes to an inconvenient language feature, but it helps until better syntax and functionality comes available.)

4 Continuations

Continuations are the second core type in AFL (together with (3.2) frames).

4.1 What The Heck Is A Continuation?

The easiest way of thinking about a continuation is that it is a function that doesn't define a return – you've got to provide an explicit exit and invoke the exit yourself. And that exit is itself is typically via a continuation.

A continuation has the following properties:

- A code location to which control is passed when the continuation is invoked.
- An assocated environment of name/value bindings. In AFL, when you transfer control to a continuation, the environment in which it was created is reestablished.
- A passed value. When you transfer control to a continuation, you pass a value – an argument list if the continuation is a function, a result if the continuation is a return, etc. "A value", because it is passed a composite value (a record) if multiple values are to be passed.
- A passed exception handler. Which could be part of the passed value. But because exception handling is in part "under the covers", the current exception handler needs to be passed explicitly, allowing the AFL run-time to know where it is. (An exception handler is itself a continuation.)

The thing that distinguishes a continuation from a "go to" is its environment (or context).

4.2 Continuations in AFL

There are two different forms of continuation specification in AFL:

- the function-like form, and
- the **catch/throw** form.

The function-like form looks like a function in the way it is used, excepting only that a coroutine's argument list is surrounded by '{ ... } rather than { ... } or (...). The distinct notation "'{" is there to tell the AFL compiler and the reader that the transfer of control does not automatically establish a return point, the way that a function call does, but rather defines and instantiates a one-way transfer of control. Here's a simple example of a function-like continuation:

```
def myloop '{n} :
(
  print n;
  myloop '{n + 1}
)
```

"`myloop`" is defined to be a continuation with an argument "`n`". It prints the value of "`n`" and then exits by invoking itself with a new value for "`n`".

The "continuation" in the '{ ... } form is the body of code following the ":". In the example, "`myloop`" is bound to the continuation value and processing continues until "`myloop`" is invoked. At that point the code in the body of the continuation is performed.

Of course, the problem with this example is that it runs forever. To deal with that, we need a way of exiting out of the loop altogether. For that we can use the other form of continuation definition, **catch**, as in:

```
catch exit :
{
  def myloop '{n} :
  (
    if n >= 10 then
      exit '{};
    print n;
    myloop '{n + 1}
  );
  myloop '{0}
}
print "OK";
```

In this second example, **catch** defines a second continuation, which is bound to the name "`exit`". A **catch** performs its body immediately. The continuation for a **catch** is the code immediately following the body of the **catch**. In the example, this is the "`print "OK"`".

A **catch** defines a name – for example, "`exit`" – which is bound to the continuation which returns from the **catch**. The continuation takes an argument value which becomes the value returned by the **catch**.

There is only one kind of continuation in spite of there being two forms of definition. A continuation is invoked using the "one-way" call, with or without a value. In the above example:

- The initial and repeating invocations of "`myloop`" pass a value to the continuation.
- The terminating invocation of "`exit`" doesn't pass a value. AFL fills in the value of **nil**, which is returned from the **catch**. But the "'{}" is required, to indicate that an invocation is to be performed.

Note that in some sense, the two forms of continuation specification are "inside out" from each other: the continuation in the function-like form is the body of the form, whereas that of the **catch** form is what follows the body.

In some programming languages, such as Ruby[9], continuations are "one-shot": once you've called a continuation you can't call it again. Amongst other things, this allows copying the program state to be done just once. Not so AFL of course: AFL goes with the more general model – allowing a continuation to be invoked any number of times.

4.3 Pluses and Minuses of Continuations

There are advantages to having continuations in a programming language:

- They help explain a lot of basic concepts in programming languages.
- They help expressing some important computing algorithms.
- They are the foundation for the implementation of many higher-level programming language functionalities.

There are also major disadvantages to having continuations in a programming language:

- They are very low-level and their use can easily obscure a program's structure. It's almost always the case that higher-level forms do the job and are more expressive of intent.

Continuations are definitely to be "considered harmful" in the same way as is the **goto** of early programming languages. But in the same way that **goto** was useful in the '70's in explaining the simple conditional and loop structures of that time, continuations are useful in explaining the "postmodern" ('though themselves dating from the '60's) control structures of AFL.

- The exposure of continuations introduces some interesting and unexpected behaviour. Continuations allow transfer of control from anywhere to anywhere else in a program. It means that function calls can return more than once per call, for example.
- There's a tendency to start with continuations when explaining other new processing models. That's quite right from a technical point of view: build the foundation first, and the house on top of that. But programmers want tools they can use and most quite rightly have no interest in such an approach.

  This document attempts to discuss higher-level things like functions, coroutines and generators first, and then explain later, in this chapter, the role of continuations. With this approach, I hope that more people can get a deeper understanding of how programming languages work.

4.4 Implementing With Continuations

4.4.1 Loops

Loops typically involve two continuations: "do it again" and "end looping". This is the definition of the **while** ... **do** loop:

```
def while [90 a ()] do [90 b ()]:
  catch exit:
    {
      def loop '{} :
        {
          if a () then
            {
              b ();
              loop '{}
            }
          else
            exit '{}
        };
      loop '{}
    };
```

There are two continuations here, "`loop`" and "`exit`", each doing what their names imply:

- "`loop`" defines one iteration of the while loop. Its implementation contains the while test. If true it evaluates the while body and reinvokes itself. If false it exits.
- "`exit`" defines where to return to when the loop terminates.

Neither "`loop`" nor "`exit`" actually pass a value. Hence the empty argument specifications.

Both the test and loop body are implemented as functions, because they both need to be invoked each time around the loop.

4.4.2 Functions

Functions are converted to an equivalent continuation form by the AFL compiler. The following two definitions are equivalent:

```
def f {A} : E ;

def f '{AA} :
{
  def A : AA . 0args;
  AA . 0return '{ E }
}
```

As are the following two calls:

```
f {V}

catch return :
  f '{ { def 0args : V ; def 0return : return } }
```

That is, a function is a continuation – the continuation that transfers control into the function's body code – and function is passed a frame of two fields:

- `0args` is the argument or arguments passed to the function, and
- `0return` is the continuation that returns from the function.

A function call uses a **catch** to establish the returning continuation.

Where a function has an argument list, it is passed as a tuple, making the following two calls equivalent:

```
f (x, y)

catch return :
  f '{ { def 0args : '(x, y) ;
         def 0return : return } }
```

and the following two definitions equivalent:

```
def f (a, b) : E

def f '{ AA } :
{
  def a : AA . 0args . 1;
  def b : AA . 0args . 2;
  AA . 0return '{ E }
}
```

4.4.3 Coroutines

Coroutines aren't a core functionality in AFL, but are defined using its facilities:

```
def makecoroutine (topFn, exit):
  {
    def c1 :
      {
        def return;
        def resume '{resumeinfo} :
        (
          return = resumeinfo.0return;
          (*c2.return) '{resumeinfo.0args.1}
        )
      };
    def c2 :
      {
        def return = '{secondArg} :
                 exit '{topFn (resume, secondArg)};
```

```
      def resume '{resumeinfo} :
      (
        return = resumeinfo.0return;
        (*c1.return) '{resumeinfo.0args.1}
      )
    };
  c1.resume
};
```

(This isn't the version of `makecoroutine` that appears in "`afldefs.afl`", but is functionally equivalent, apart from the maintenance of per-coroutine `currentTextWriter`, `currentTextReader` and `currentSource`.)

The most important features of this implementation are:

- Two "portals" are defined so as to refer to each other. The "`resume`" method of each portal transfers control to the other coroutine. The two portals are symmetric in all respects except that one of the two starts the other one running. The starter is referred to as the primary coroutine and the startee the secondary coroutine.
- Each portal's resume method is implemented as a continuation that acts like a function from the point of view of its invoker. From the other coroutine's point of view the passed argument value is returned as the result of invoking its resume method.

  What `resume` does is save away its return address, and then returns using the return address of the other coroutine. Apart from that, all it does is return the argument value passed it.
- The "`return`" of the secondary portal is set to invoke the "`topFn`" passed to makecouroutine, and to invoke "`exit`" on its return. The first time the secondary coroutine is resumed – is returned to – it invokes `topFn`.
- `makecoroutine` returns the `resume` function of the primary portal.

The reason continuations are used in the implementation of coroutines is that they give explicit access to the return continuations of the coroutine's "`resume`" methods.

4.5 Continuation Syntax in More Detail

There are two kinds of continuation definition, and syntactic variations in both cases:

- There are three forms of the function-like definition in terms of how they specify or not an argument name:

  ```
  '{name} :
  '{} :
  '{contArg} :
  ```

  The **contArg** keyword can be used within a '{}:'s body. It provides access to the argument value passed to the continuation and can be used either in the named form or unnamed form. The "'{contArg}:" specification makes this clear but isn't required.

  contArg needs great care in its use. Because of how the AFL compiler rewrites higher-level functionality, defining and using functions, operators and catch can affect the value, not only within their arguments and insides, but in following code. To avoid this difficulty, avoid using `'contArg` explicitly: use the named form if the passed value is used, and use ('{}) if the value isn't used.
- There are also three forms of **catch** in terms of how they specify or not an exiting continuation name:

  ```
  catch name :
  catch :
  catch throw :
  ```

  There's no ('{}:) used, because the name is not the argument of a continuation, its a continuation itself.

  The **throw** keyword can be used within a **catch**'s body. It provides access to the to the continuation and can be used either in the named form or unnamed form. The "catch throw:" specification makes this clear but isn't required. For example, an alternative formulation of a function call in terms of continuations is the following:

  ```
  catch :
    f '{ { def 0args : V ; def 0return : throw } }
  ```

  Like `contArg`, explicit use of `throw` should be avoided == it's used in AFL1-to-AFL0 rewrites.

Like functions, function-like continuations can be defined and bound to a name with a variant form of the **def** declaration. These two examples are equivalent:

```
def loop '{} : (b (); loop '{});
def loop : '{} : (b (); loop '{});
```

The main purpose of the **contArg** and **throw** keywords is to move all naming issues into frames. The AFL compiler rewrites the named forms of '{} definition and **catch** into frames and the non-name forms. It converts:

```
'{ A } : E
```

into

```
'{ } : {def A : contArg; E }
```

and

```
catch exit : E
```

into

```
catch : { def exit : throw; E }
```

4.6 Exceptions and Continuations

(2.12) Exception Handling describes the **try** ... **except**, **try** ... **finally**, **signal**, **resignal**, **returnFrom** and **return** facilities for handling implicit non-local exits. These facilities are implemented using lower-level functionality.

The current exception handling functionality doesn't distinguish between different kinds of exceptions – an exception handler gets all exceptions – although you can ask the type of an exception and handle it or pass it on accordingly.

4.6.1 The Current Exception Handler

AFL maintains a "current exception handler", which is invoked when a .NET exception happens. The current exception handler is accessible using the keyword **exceptionHandler**:

```
except (e):
  if typeOf e != "DivideByZeroException" then
    exceptionHandler '{e};
```

The value of **exceptionHandler** is a continuation, hence the '{ ... } invocation syntax. It can be assigned and passed like any other value, not just invoked in the above manner. The value passed to it is a .NET exception.

The value of **exceptionHandler** with an exception handler is always the the next outermost exception handler. In the above example, using `exceptionHandler` invokes the next outer current exception handler – it does the job of **resignal**.

4.6.2 Establishing Exceptions

Each continuation has an associated exception handler that is bound to it when the continuation is created. Whatever the current exception handler is when a continuation is created becomes that continuation's associated exception handler. When a continuation is invoked, its associated exception handler becomes the current exception handler.

There are basically two cases for establishing a current exception handler:

- One generally wants called functions to be evaluated in the context of the caller's exception handler rather than the current exception handler at the time the function was defined.
- One generally wants the exception handler of the calling context to be restored when a function is returned from.

The return case is well handled by the default behaviour of continuation invocation: reestablish the continuation's associated exception handler. The call case requires something more. This case is dealt with by an alternative form of continuation invocation, that has two arguments:

```
myCont '{argument, exceptionHandler}
```

The continuation is invoked with given argument as in the single-argument continuation invocation case. In addition, the current exception handler is set to the value given as the second argument, rather than that associated with the invoked continuation. In this example the used exception handler is **exceptionHandler**, the current exception handler at the time of the call, which is the case when calling functions.

The rewrite that converts function calls to continuation invocations uses this two-argument form of invocation. The following two calls are equivalent:

```
F {A}
```

```
catch exit:
  F '{ {def 0args: A; def 0return: exit},
       exceptionHandler }
```

There's a corresponding explicit exception handler passing version of function call:

```
F {Argument, exceptionHandler}
```

This is the non-argument-list form of function call. With two arguments, the first is passed to the function, and the second becomes its current exception handler. Where there is an argument list, it can be passed in this form as a tuple:

```
F { '(Argument1, Argument2, Argument3),
     exceptionHandler }
```

4.6.3 Every Continuation Has An Exception Handler

In addition to there being an always accessible current exception handler, each continuation has an associated exception handler which can be accessed by using the keyword **exceptionHandler** as if it were a field name:

```
exit . exceptionHandler
```

The primary use of qualified **exceptionHandler** is to explicate the rewrite of:

```
exit '{}
```

as:

```
exit '{ {}, exit . exceptionHandler }
```

4.6.4 Resignal and Signal

resignal is implemented by a simple invocation of the current **exceptionHandler**:

```
def resignal [180 e]: exceptionHandler '{e};
```

signal is similar, with the addition of a call to the .NET method for creating new Exception object instances:

```
def signal [90 a]:
  exceptionHandler
   '{"new void System.Exception(string)" (repr a)};
```

4.6.5 try/except and try/finally

The two forms of **try** are implemented using two-argument continuation calls.

try ... **except** invokes its **try** part with its **except** part as its **exceptionHandler**:

```
def try [90 c ()] except [90 e]:
  catch exit:
    c '{{def 0args: '(), def 0return: exit},
        '{o}: e '{{def 0args: '(o),
                   def 0return: exit}}};
```

Both the **try** and **except** parts have the same return location: exiting from the **try**. Which is why the **try** part is invoked with the same `0return` value as is used to invoke the **except** part.

try ... **finally** is implemented similarly:

```
def try [90 c ()] finally [90 e ()]:
  {
    def r: catch exit:
          c '{{def 0args: '(), def 0return: exit},
              '{o}: (e (); exceptionHandler '{o})};
    e ();
    r
```

```
};
```

The major features of the **finally** form are:

- The result of the **try** part is saved (as "r") and its value returned in the "normal" logic path.
- The **finally** part is performed in both paths out of the form: in the "normal" path immediately prior to returning the result value, and in the exception handler used by the **try** part.

4.6.6 returnFrom and return

To support **returnFrom** and **return**, the AFL runtime supplies a special .NET Exception derived type: AflReturn. This exception type is issued by **return**, trapped by **returnFrom**, and is used to pass the result value:

```
def returnFrom [90 c ()]:
  try
    c ()
  except (e):
    if typeOf e == "AflReturn"
    then o.Value
    else resignal e;

def return [90 v]:
  exceptionHandler
    '{"new void AflRuntime.AflReturn(object)" (v)};
```

returnFrom is most simply defined in terms of **try/except** and **resignal**. In "afldefs.afl", it's defined in lower-level terms.

4.7 atEnd vs. finally

atEnd and **try/finally** appear to be similar, but the effect of their use is quite different:

- One can't be entirely sure when an **atEnd** is performed, but one can be sure that it will be performed. An **atEnd** is not performed at the end of the expressions in its frame – it is preformed when the memory used by the frame is reclaimed by the run-time system, which may be at a much later time.
- One can't be sure that a **finally** will be performed, but one can be sure when it will be performed in the flow of program logic. A **finally** is performed when the code in its **try** part exits with a value, or with an exception. However, in AFL, this may never happen. As a rather artificial example:

```
catch exit:
{
  try
    exit '{}
  finally
    print "This will never be printed."
}
```

4.8 Callability

In (3.2.6) Calling Frames, it is described how to make a frame callable using the "0callable" property. In that section, the value of the "0callable" property is described as a function or frame. In practice, it's a continuation or a frame: it can be a continuation.

4.9 Stackless Programming

AFL is a "stackless" programming language. That is, no stack is used in the evaluation of expressions. (Actually you've got to use .NET's run-time stack to communicate with the .NET libraries and use .NET run-time operations, but that's the only use AFL makes of the stack.) A stackless model makes it easy to implement the functionality found in AFL. There are other implementation techniques for much of that functionality, but totally getting rid of the stack opens up all sort of possibilities.

Without a stack, all intermediate results (most importantly when invoking continuations) need to be placed somewhere else, typically in frames. Function argument lists are combined into a single frame which becomes the single argument passed to a continuation. And the three values required to invoke a continuation – the continuation value, the argument value, and the passed exception handler – need to captured likewise. To do this requires a low-level form of continuation call, which accepts the bundled values:

```
ccallWithEH '(cont, arg, exc)
```

The **ccallWithEH** operator expects a frame as its argument, with its "1"-named field the continuation value, its "2"-named field the passed argument and its "3"-named field the passed exception handler.

5 Other Functionality

Useful examples of functionality that can be accessed and implemented using AFL in its current state are text input and output, pattern matching and threading. (Mostly because text processing applications are those that I find most interesting, and to make it easier to compare coroutines and asynchronous threads.)

It's informative to look at the include files "afldefs.afl", "io.afl", "patterns.afl", "sourceio.afl" and "threading.afl". The following descriptions should be sufficient for using these libraries, but for what's really going on, the include files are the authoritative source.

5.1 Odds and Ends

The following functionality is defined in "afldefs.afl" – which is used by default – but hasn't been explained elsewhere.

5.1.1 Time

`now`

> "`now`" is a no-argument operator that returns the current date and time. The returned value is a .NET value, but if you apply the "`repr`" string operator to its value or print it, it will give a formatted representation of the date and time.

`expr1 - expr2`

> Both expr1 and expr2 must be .NET date/time values of the sort returned by `now`. The `-` operator returns a .NET time interval value. If you apply the "`repr`" string operator to its value or print it, it will give a formatted representation of the time difference.

5.1.2 Done

`done '{ programResult }`

"`done`" is a continuation that terminates the current thread or the main program. For the main program, if the argument passed to it is a string value that string value is printed, and if the argument is a number its value is used as the program's exit code. In all other cases, if the argument is not a string or number, or if the exit is from a thread other than the main program, the argument value is ignored.

5.1.3 Reflection

`typeOf expr`

"`typeOf`" is an operator that returns a string description of the type of its argument expr.

`expr1 hasField expr2`

"`hasField`" is an operator that returns true if expr1 is a frame or other .NET object with a field named expr2. expr2 must be a string-valued expression.

`expr1 field expr2`

"`field`" is an operator that returns the field of a frame or other .NET object expr1, whose field name is expr2. expr2 must be a string-valued expression.

5.1.4 Random Numbers

`random (number)`

`random` returns a uniformly distributed non-negative number with a value less than number.

5.1.5 Building Continuation Call Argument Lists

`ccall tuple`

Assumes tuple is a tuple of length 2, with the first item a continuation and the second its argument. Calls the continuation with the argument and the current exception handler.

The following continuation calls are all equivalent:

```
C '{A}
C '{A, exceptionHandler}
ccall '(C, A)
ccallWithEH '(C, A, exceptionHandler)
```

5.2 I/O

The include file "io.afl" defines a number of useful values and functions for reading and writing text, and for reading and writing octet/byte data.

5.2.1 Text Output

A "`textWriter`" is an AFL wrapper for a .NET System.IO.TextWriter value. Its methods are:

`textWriter.writeline (text)`

writes a line containing the text text.

`textWriter.write (text)`

writes the text text, no following line-end.

`textWriter.flush ()`

flushes out the text previously written.

`textWriter.close ()`

flushes and then closes the .NET System.IO.TextWriter.

Functions, operators and values using `textWriter` are:

`stdout`

is a textWriter that writes to "standard output".

`stderr`

is a textWriter that writes to "standard error".

`outputFile (fileName)`

is a function that creates a `textWriter` to the named file.

`textWriter (NETTextWriter)`

is a function that wraps a .NET System.IO.TextWriter.

5.2.2 String I/O

For convenience, string buffers can be written to:

`stringWriter`

is a no-argument operator that returns a text writer that writes to a string buffer. You can retrieve what's been written using the `readback` method:

`textWriter.readback ()`

retrieves what's been written to a text writer as a string value.

5.2.3 Text Input

A "`textReader`" is an AFL wrapper for a .NET System.IO.TextReader value. Its methods are:

`textReader.readline (exit)`

reads and returns a line of text.

`textReader.readchar (exit)`

reads a character, returning its value as an integer.

`textReader.peekchar (exit)`

reads a character, returning its value as an integer, but does not consume it, so that the same character is read next time.

`textReader.readtoend ()`

reads in and returns all the remaining text in the file.

`textReader.read (needed)`

reads in at least needed characters in text mode, reading in multiple lines if need be. A line feed is added to the end of each read line by `read`.

textReader.generate

returns a generator of the input's text lines. A textReader can therefore be used as the first argument of "for each". For example:

```
for each stdin do (line):  # Print out all non-
  if line != "" then       # empty lines from
    print line;            # standard input.
```

Each of "readline", "readchar" and "peekchar" takes a continuation as an argument and transfers to that continuation when end of input is encountered.

Functions and values using textReader are:

stdin

is a textReader that reads from "standard input".

inputFile (fileName)

returns a textReader that reads from file fileName.

textReader (TextReader)

is a function that wraps a .NET System.IO.TextReader.

5.2.4 The Current Text Reader and Writer

At all times, "io.afl" keeps track of a "current text reader" and a "current text writer". These values are initialized to "stdin" and "stdout", but they can be changed by assigning to them:

```
currentTextReader = inputFile ("myinputfile.txt");
currentTextWriter = outputFile ("myoutputfile.txt");
```

The "writeline", "write", "flush", "close", "readback", "readline", "readchar", "peekchar" and "readtoend" methods can be used without qualification when used with the current text reader or current text writer:

```
writeline ("Hello World");
catch exit:
  {
    def line : readline (exit);
    # some processing of line
  }
```

5.2.5 Text Writing Operators

There's an operator that can be used as a prefix or infix that makes writing to the current text writer a bit easier:

<< text

is an operator that writes the text text to the current text writer, and which returns the current text writer as its result.

textWriter << text

is an operator that writes the text text to the given textWriter, and which returns the textWriter as its result.

The prefix and infix forms of << can be chained as in:

```
<< "Hello World" << lf
```

5.2.6 Text I/O Localization

Assigning to currentTextWriter and currentTextReader can been clumsy, especially if there is a "main" output or input, as well as more localized need for wanting to use the module's facilities. There's a few control forms that help:

withTextWriter textWriter do expression

withTextWriter saves away the current value of currentTextWriter, sets it to textWriter, evaluates expression, restores the saved value of currentTextWriter, and returns the value of the expression. withTextWriter allows local unqualified use of the writing functionality without disrupting a more global current text writer.

withTextReader textReader do expression

withTextReader saves away the current value of currentTextReader, sets it to textReader, evaluates expression, restores the saved value of currentTextReader, and returns the value of the expression. withTextReader allows local unqualified use of the reading functionality without disrupting a more global current text reader.

readFrom expression

readFrom saves away the current value of currentTextWriter, sets it to a stringWriter, evaluates expression, restores the saved value of currentTextWriter, and returns the text written to the stringWriter as a result.

readFrom differs from streamFrom in that it collects the string value and when finished returns its value, whereas streamFrom returns text as it is written within streamFrom.

5.2.7 Text I/O Between Coroutines

File readers and file writers are essentially coroutines – they are synchronous but separate processing threads. Reading and writing is also useful as a way of communicating within a program: data is "piped" through subprocesses. The streamFrom and streamInto operators support this form of programming.

streamFrom expression

streamFrom returns a textReader that supports all the methods of a (5.2.3) text input. The data to be read is whatever is written to currentTextWriter within expression.

expression is performed in a separate coroutine, whose initial currentTextReader and currentSource are set to those of the invoker of the streamFrom. Its initial currentTextWriter is created so that anything written to it is returned when the streamFrom textReader is read from.

streamInto expression

streamInto returns a textWriter that supports all the methods of a (5.2.1) text output. The data written is read by the currentTextReader within expression.

expression is performed in a separate coroutine, whose initial currentTextWriter and currentSource are set to those of the invoker of the streamInto. Its initial currentTextReader is created so that anything written to the streamInto textWriter is made available on following reads from the currentTextReader.

5.2.8 Text Filters

A text filter is an expression (typically encapsulated in a function) that reads from its default input and writes to its default output, transforming the data in some manner. A filter can be used standalone, as the main logic of a program. Or it can be used as a "pipe" within an AFL program. There are two operators that support attaching text filters to readers and writers:

`expression *> textWriter`

> The expression is evaluated with its `currentTextWriter` set to the given `u(textWriter). The result of the `*>` operator itself a text writer, such that anything written to it is available to be read within the expression using the `currentTextReader` of that expression.

`textReader <* expression`

> The expression is evaluated with its `currentTextReader` set to the given `u(textReader). The result of the `<*` operator itself a text reader, such that anything written to the `currentTextWriter` of the expression is available for reading from it.

The results of `*>` and `<*` can be used in any writer or reader context. Multiple uses of `*>` or multiple uses of `<*` can be used to pipe multiple filters together. A filter can be used in an input or an output context: a filter need not be written specifically for input or output. For example:

```
someReader <* filter1 <* filter2   # a reader
filter1 *> filter2 *> someWriter   # a writer
```

5.2.9 Print

The "`print`" operator always writes to "`stdout`". It is an operator, with a low-precedence argument, so it can be used as if a "statement":

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

"`printinline`" prints to "`stdout`" like "`print`", but it doesn't put a line-end at the end of what it prints.

`print` and `printinline` are defined in "`afldefs.afl`" rather than "`io.afl`", so they're always available, even if "`io.afl`" isn't included in the program.

5.2.10 Octet Input

Not everything is text. To aid in using non-text data, "io.afl" provides low-level functionality for octet/byte data. For octet input:

`octetInputFile (fileName)`

> Open a file for octet input. "`octetInputFile`" returns an "`octetInputStream`" that can be read from.

`octetInputStream.read (needed)`

> Read the specified number of octets from octetInputStream. ".read" returns a string. If there are fewer than needed octets left in the input, all the remaining octets, if any, are returned.

5.2.11 Octet Output

For octet output:

`octetOutputFile (fileName)`

> Open a file for octet output. "`octetOutputFile`" returns an "`octetOutputStream`" that can be written to.

`octetOutputStream.write (data)`

> Write the specified number of octets to octetOutputStream. (Only the bottom 8 bits of the Unicode number of each character in data are written.)

`octetOutputStream.close ()`

> Close the octetOutputStream. A ".close" ensures that all the data written gets to its destination. (`octetOutputStream` closes itself, but at an unspecified time following its last use, which may be too late if the output stream is reread during the running of the program.)

5.3 Pattern Matching

The include file "patterns.afl" contains functions and operators that perform Icon-like pattern matching.

There are three components to textual pattern matching:

- the scannable source of text to be matched,
- the patterns that describe what's to be matched, and
- the mechanism for applying a pattern to a scannable text source.

The mechanism is the easy part in AFL. Pattern matching operations return a true/false result, indicating whether the pattern matched or not. The scannable text source has properties that indicate what it was that was matched.

This implementation is a non-backtracking pattern matching model, which works well for a large range of tasks. "Non-backtracking" means that when an alternative fails, its components are not revisited for sub-alternative possibilities.

5.3.1 Scanning Sources

There are a number of ways of creating a scannable text source:

`scanningSource (inputFunction)`

> "`scaningSource`" takes a function as its argument. "`inputFunction`" is expected to be a function that takes a numeric argument. When called, inputFunction should return at least that many characters of text (it can return more than requested), or return fewer characters when at the end of input.

`stringSource (string)`

> "`stringSource`" takes a string and makes it into a scannable text source.

`fileSource (textReader)`

> "`textSource`" takes a textReader (as defined in "io.afl") and

makes it into a scannable text source.

`currentSource`

"`currentSource`" is the scannable text source used in the most recent pattern match. `currentSource` is a variable, so needs dereferencing when the source value is needed. However, most user uses of `currentSource` are assignment to, and most value accesses are inside "patterns.afl". For example:

```
currentSource = fileSource (stdin);
```

`scanReader`

`scanReader` returns a scannable text source that reads from the `currentTextReader`. It's provided as a short-hand for the somewhat clumsy:

```
scanningSource ((*currentTextReader).read)
```

`scanReader` is most useful when combined with source-establishing operations, such as:

```
withSource scanReader do expression;
currentSource = scanReader;
```

`scanReader` needs "io.afl" as well as "patterns.afl" to be included to be useful.

`matched`

"`matched`" is a no-argument function that returns the value of the current scanning source.

5.3.2 *Applying Patterns to Scanning Sources*

Each of these operators apply the pattern to the scanning source. If the scanning source is a string value, it is promoted to a scanning source for pattern matching. The difference between the operators is the result they return.

`scanningSource ~: pattern`

Return true or false.

`scanningSource +: pattern`

Return the text that's matched.

`scanningSource -: pattern`

Return all the text following what's matched.

The two-argument forms of `~:`, `+:` and `-:`, each bind their first argument value to `currentSource` while the pattern is being matched, and restore the previous value at the end of the match. In general, if the user wishes to recover properties of the scanned source after a match, the source should be captured in some other manner.

`~:`, `+:` and `-:` can also be used as prefix operators, with just a pattern as their following argument. In these cases, `currentSource` is the scanning text source matched against. It's very often the case that it's appropriate to assign your input to `currentSource` and then use it implicitly from there on in.

`withSource scanningSource do expression`

`withSource` saves away the current value of `currentSource`, sets it to scanningSource, evaluates expression, restores the saved value of `currentSource`, and returns the value of the expression. `withSource` allows local uses of scanning, while a more global use of a scanning source is going on.

scanningSource can be a scanning source value, a string or a text reader value (as created in "io.afl"). `withSource` detects which it is and convert is to a scanning source as appropriate.

5.3.3 *Patterns*

`succeed`

Always succeed matching, consuming no input text.

`fail`

Always fail to match.

`= string`

Match the given string of text.

`pattern |: pattern`

Match the first pattern. If it doesn't match, match the second pattern.

`pattern &: pattern`

Match both patterns in the order given.

`pattern :?`

Match zero or one instance of pattern.

`pattern :*`

Match zero or more instance of pattern.

`pattern :+`

Match one or more instance of pattern.

`pattern rep count`

Match pattern exactly count times. "`rep`" fails if count is negative.

`pattern rep count orMore`

Match pattern count times or more. "`rep 0 orMore`" is equivalent to "`:*`", and "`rep 1 orMore`" is equivalent to "`:+`".

`pattern repupto maxcount`

Match pattern (at least zero but) no more than maxcount times.

`pattern rep count upto maxcount`

Match pattern at least count times but no more than maxcount times.

`anyOf string`

Match any one character so long as it's in the given string.

`noneOf string`

Match any one character so long as it's not in the given string.

`any`

Match any one character.

`?: pattern`

Match the given pattern, but don't consume any input.

`!: pattern`

Fail if the given pattern matches and succeed if it doesn't. Don't consume any input either way.

`arb pattern`

Match up to but not including the given pattern. Fail if the pattern is not found.

`arbplus pattern`

Match up to but not including the given pattern. Fail if the pattern is not found, or if zero characters are matched prior to the pattern.

`move number`

Consume number characters. Fail if there aren't that many characters, and leave any examined characters in the input for later use.

`pos number`

Succeed if number is the current character position in the input. For non-negative values of number, 0 is the start of the input. For negative values of number, -1 is the end of the input.

`pattern :> string`

Assign the text matched by the pattern to the scanning source's saved string property named by the string.

`another string`

Match the text most recently assigned to the scanning source's saved string property named by the string. Fail if the saved string doesn't match or if there is no such saved text.

`character .. character`

Match one character, so long as its "ord" value is greater than or equal to that of the first character and less than or equal to that of the second character. Fails if the next character in the input is out of range or if there are no more input characters.

5.3.4 Accessing Scanning Source Properties

Once a pattern has been matched, a scanning source can provide information about what was matched.

`source []`

returns all the text matched by the most recent pattern match on source.

`matched []`

returns all the text matched by the most recent pattern match on the current source.

`source [string]`

returns the text most recently assigned within a pattern (by `:>`) when scanning source.

`matched [string]`

returns the text most recently assigned within a pattern (by `:>`) when scanning the current source.

5.4 Scanning Text Readers

"sourceio.afl" defines a few operators that combine the functionality of text readers from "io.afl" and the pattern matching functionality of "patterns.afl". Both these latter files must be included if "sourceio.afl" is. (AFL doesn't yet support include dependencies.)

`textReader <~ expression`

supplies the textReader as the current scanning source of the expression, and returns a text reader that reads what's written to the current text writer of expression.

`scanFrom expression`

returns a scanning source whose input whatever's written to the current text writer of the expression.

Both these operations are readily definable in terms of other functionality, but provide useful expressive forms in programs whose primary processing is pattern-matching-based.

5.5 Threads

The include file "threading.afl" contains functions that support asynchronous threading and simple synchronization.

5.5.1 Creating and Using Threads

`thread (function)`

"`thread`" creates and returns a new thread. The new thread starts by invoking the passed function with a zero-length argument list. The new thread runs until it exits from function. A thread is created in a non-started state. It needs to be started by calling its ".start" method.

`currentthread`

"`currentthread`" is a no-argument operator that returns the thread in which its invoker is running. In the same way that all code is running in a coroutine, all code is running in a thread. No attempt should be made to ".start" `currentthread`.

The value returned by `thread` or `currentthread` is a record of five function-valued fields (or methods):

`thread.start ()`

Start the thread. Only call this once.

`thread.abort ()`

Abort the thread.

`thread.join ()`

Suspend the invoking thread until thread terminates.

5.5.2 Monitors

`monitor`

"monitor" is a no-argument operator that creates and returns a new synchronization monitor. A monitor has the property that only one thread at a time can own it. The returned value is a record of two function-valued fields (or methods):

monitor.enter ()

".enter" acquires ownership of the monitor for the thread that invokes ".enter". If another thread attempts to acquire ownership of the same monitor, it will block until the owning thread does an ".exit". If another thread already owns the monitor, the invoking thread waits until the monitor becomes available.

A thread can call ".enter" more than once. The thread retains ownership until it calls ".exit" as often as it has called ".enter".

monitor.exit ()

".exit" releases ownership of the monitor. If multiple ".enter" calls have been made, ownership is retained until there has been an ".exit" for each ".enter".

critical monitor do expression

Evaluate and return the result of expression, wrapping the evaluation in an ".enter" and ".exit" on monitor. critical ensures that the ".exit" is performed no matter how expression terminates, making it a lot safer than using ".enter" and ".exit" oneself.

5.5.3 Sleep

sleep (milliseconds)

Puts the invoking thread to sleep for the indicated number of milliseconds.

5.5.4 Multi-Thread Variables

Variables of the sort described in present a problem when used in multi-threaded applications: use of a variable in more than one thread can produce unpredictable results – two sequences of operations can be interleaved in arbitrary ways. There are three ways of dealing with this difficulty:

- Provide each thread with its own variables, preferably declared in mutually disjoint scopes, so one thread can't see the variables of the "main" program or another thread. This is the best approach for most purposes.
- Use (5.5.2) monitors to lock the access to one or more variables. This is the best approach where there really is just one variable in question, shared by all threads.
- Use multi-thread variables to provide each thread with its own value. Multi-thread variables allow a global name to refer to a thread-specific value, allowing easier implementation of implicitly used varibles such as currentTextReader, currentTextWriter and currentSource.

A multi-thread variable, or thread-safe reference, acts like any other variable – you can get and set its value – with one important difference: each thread has its own separate value for the variable. Even if you pass the variable between threads, any use of the variable remains specific to the thread in which the use is made. A multi-thread variable is created using the **threadSafeRef** operator:

threadSafeRef (initialValue)

Creates a new multi-thread variable with an initial value of initialValue. The initialValue is the initial value of the variable in each thread in which it is used. Assigning to the variable in one thread only changes the value in that thread.

A named thread-safe variable definition binds the result of **threadSafeRef** to the name as follows:

```
def x : threadSafeRef nil;
```

The corresponding non-thread-safe variable definition would be:

```
def x : ref nil;  # or
def x = nil; # or
def x;
```

5.5.5 currentTextReader, currentTextWriter and currentSource in Threads

As noted earlier, currentTextReader, currentTextWriter and currentSource are thread safe. This is implemented by making these values multi-threaded variables.

The initial values of each of these variables is **nil** in all threads except for the main program (where they are set to other values if "io.afl" or "patterns.afl" are included). So these values need to be set if they are to be used, or if the I/O or pattern matching operations that use them implicitly, in a thread.

5.6 XML Parsing

The include file "xmlparser.afl" provides an interface to .NET's Validating XML Parser. This interface doesn't provide all the functionality of .NET's XML Parser, but it provides a useful subset and is an example of connecting to external functions and types.

A new XML Reader is created using the xmlreader function:

xmlreader (namepaces, xmlFragment)

xmlreader creates a new .NET XML Reader and passes it the given namespace mappings and fragment of XML text.

The namepaces must be a hashtable, with the key/value pairs each being a namespace prefix (the key) and its URN mapping (the value). If there are no such mappings, you can use "hashtable ()" as the first argument, which is recognized with the same meaning. Mappings can be specified in the first argument value using the '[...] hashtable notation:

```
def reader : xmlreader ('["bk", "urn:sample"],
          <book>
            <title>Pride And Prejudice</title>
            <bk:genre>novel</bk:genre>
          </book>
        );
```

The xmlFragment must be a string value.

xmlreader.read ()

".read" attempts to read in the next XML node. It returns true or false, indicating whether one has been read.

xmlreader.nodeType ()

".nodeType" returns the type of a read-in node. You can

"repr" this value to get a string representation of the type, such as "Element".

xmlreader.depth ()

".depth" returns the depth of the node in the structure of the parsed XML fragment.

xmlreader.localName ()

".localName" returns the local name of the node – e.g. an element name.

xmlreader.prefix ()

".prefix" returns the prefix of the node. For example, the "xsd" in "xsd:path". ".prefix" returns the zero-length string if there is no prefix.

xmlreader.namespaceURI ()

".namespaceURI" returns the namespace associated with node's prefix, if any.

xmlreader.value ()

".value" returns the string value of a node with a text property.

xmlreader.attributes.generator (yield)

".attributes.generator is a generator of the node's attributes. The generator yields for each attribute, its prefix, attribute name and value, as in:

```
for each xmlreader.attributes do
  (prefix, name, value):
    if prefix == "" then
      printinline " \{name}=\"\{value}\""
    else
      printinline
               " \{prefix}:\{name}=\"\{value}\""
```

xmlreader.attributes [name]

Indexing ".attributes" with a valid attribute name returns the attribute's value.

An xmlreader can be used as a generator itself. For each generation, it reads in the next XML node and yields the name of the node's type, as in:

```
for each xmlreader do (nodeTypeName):
  if nodeTypeName == "Element" then
    # process element etc.
```

6 Implementation Issues

6.1 The Single Frame Stack Tyranny

The single frame stack technique common in current programming languages was developed in the 1960's to deal with the slow machines and small memories of the time – we're talking 8KB of memory and either no hard disk or, if you were very lucky (and rich), maybe a 256KB drive. In these circumstances, there is neither the processing power nor the available memory to do anything other than squeeze everything in as tight as possible, in as short a time as possible.

A single frame stack isn't the only optimization available. With faster processors and more memory – if say we could afford a machine with as powerful as a 10MHz processor with 2MB memory, we could have multiple frame stacks, or eliminate them altogether.

The single frame stack is the major impediment in improving current programming languages.

6.1.1 A World With No Frame Stacks

The big "secret" of AFL's functionality is that AFL doesn't have a "frame stack" – when a function is called, the memory required for its arguments and for the local state of the function are allocated on the "heap", a persistent pool of memory that requires garbage collection to recover no-longer used memory. This is something of an extreme approach, but it's appropriate for prototyping a programming language.

In practice, the AFL implementation uses .NET's run-time stack for invoking .NET functionality, and accessing values. In both cases, values are put on the stack, the operation performed, and all values immediately removed from the stack. Likewise, AFL continuations use .NET local variables, but only for immediate use, and there's only a very small finite number of them (about 4 total) in existence at any one time.

Having no frame stacks is not as radical (or even new) an approach as it might at first seem. The frame stack is essentially an optimization of a certain kind of memory allocation and deallocation. It allows very fast creation and releasing (simply moving the stack pointer), that works especially well for small pieces of memory – like an argument list or a function's local state. But if machine architectures weren't stuck in the '70's, they could provide support for alternate allocation techniques.

6.1.2 Multiple Frame Stacks

AFL goes a bit too far in the direction of stacklessness.

The major missing feature in most programming languages is coroutines and full-function generators. Completely general continuations require stacklessness. But completely general continuations are not desirable in "real" programming languages. Support for coroutines and fully general generators only needs a move away from the single-stack mode to a multiple-stack model. Current hardware architecture supports multiple stacks, but current operating systems (both Windows and Linux) make assumptions about how the hardware stack pointer is used: Linux uses the stack pointer to identify the current thread, and Windows seems to use it in low-level heap memory management. Both operating systems make the assumption that each asynchronous thread has only a single execution stack. And both make efficient implementation of coroutines unnecessarily difficult.

Backing off from the ability to use continuations in any context, but sticking with full-functioned coroutines and generators would not reduce the power of the language significantly, while providing a great deal of opportunity for code optimization, even given current operating systems' impediments in doing so.

6.2 Layers and Stages

AFL has been implemented in stages and is implemented in layers.

6.2.1 Layers

The AFL compiler presently consists of five phases:

- A lexical analyzer that reads the source files, recognizes tokens and implements "`include`" and "`externalModule`". The lexical analyzer also recognizes basic structural components such as parenthesization and semicolon and comma separation.
- A parser that builds the syntax tree out of the lexical analyzer's tokens. The parser implements operator definition and rewrites uses of operators into function calls.
- A rewriter that transforms the parser's syntax tree in a number of ways. It transforms function definitions and calls into continuations and catches, and rewrites uses of names into more basic forms of frame use.

  The result of either the parser and rewriter phase can be output with the /ah and /al options respectively. The output is in AFL in both cases.
- A "flattener" that transforms the rewritten syntax tree into a flattened sequence of continuations. The output of the flattener can be produced using the /flat option.

  Unlike the results of the earlier phases, the output of the flattener is not in AFL – it's a bit too low-level even for AFL.
- A code generator, that translates the continuations and frames into target code. At present there are two code generators: one that translates into C# and another that translates into .NET assembler language. The result in either case has to be compiled using the C# compiler (csc, msc or some other name) or CIL assembler (ilasm or some other name) into a .NET assembly that can then be run. The two approaches are equivalent in all respects except that the C# approach doesn't use a "native" implementation of continuation transfers: it uses a "trampoline" function instead – returning to the trampoline at the end of each continuation, and having it call the next continuation. The CIL implementation uses tail calls, with the same effect, but more efficiently. A compile-and-go and a compile directly to a .NET assembly code generator are in the works.

6.2.2 Stages

The AFL compiler was developed in stages. Evidence of this is in the rich output available from each phase of the compiler. The compiler has undergone a lot of changes since its start (October 2004), one of the major ones being its integration with the .NET run time libraries.

Importantly as a learning exercise, the C# code generator was created first, and difficulties in developing the CIL assembler code generator dealt with by examining the disassembly of the C#-based compiled code. It was generally easier to express intent in C#. This process is continuing in developing the direct-to-code code generator, using the CIL assembler as a guide.

6.3 Continuations

Continuations are implemented differently by the C# and for the CIL code generators. Both code generators implement continuations as .NET classes with a method (named "`_0CCall`") to be called in order to transfer control to the continuation. It's how control is passed out of the `_0CCall` method that differs between the two code generators:

- The CIL code generator implements a continuation transfer as a tail-call from within the invoking continuation. `_0CCall` methods don't return, but always transfer control with a tail-call. The .NET implementation correctly implements tail-calls, so that the call stack is only ever one level deep.
- The C# language has no way of expressing a tail-call. So instead:
  - For a top-level transfer to a coroutine, the coroutine (object) to be called is passed to a "trampoline" function ("Afl.CallContinuations") which repeatedly calls `_0CCall` methods.
  - For a tail-call, the continuation to be called is saved away and the calling continuation's `_0CCall` method simply returns to the Afl.CallContinuations that called it.

Neither of these approaches is ideal implementation-wise, but they both work. The CIL implementation is preferable performance-wise.

Asynchronous threading requires that the top-level of a thread pass control to an AFL continuation. This is achieved by having each thread top-level pass the continuation to Afl.CallContinuations – which has a useful property that if the called continuations tail-call each other, it exits its looping the first time it is returned to.

6.4 The Inheritance Problem

The keeping of separate copies of `currentTextReader`, `currentTextWriter` and `currentSource` for each coroutine requires, in the language as it stands, that the coroutine implementation be aware of these values. In a language with a behavioural inheritance system – like an object-oriented language – it would be possible to separate this functionality, but not in the current AFL language.

6.5 Temporary and Transient Values

The AFL compiler allows you to omit the rewriter phase (the /rw- command-line option). Great care needs to be taken if you do so, because the rewriter is responsible for allocating locations for intermediate values – you are programming directly in AFL0 rather than AFL1/AFL2. The values affected are:

- **self**,
- **throw**,
- **contArg**, and
- the result of evaluating a frame.

The difficulties with (3.2.4) self, throw and (4.5) contArg are discussed with their descriptions earlier. Evaluating a frame has the same difficulties. The solution, like the other cases is to bind the value of a frame to a name of an enclosing frame prior to the evaluation of any nested frame.

If you want to experiment with /rw-, it's appropriate to bind the values of **contArg**, **throw** and **self** to local names as soon as possible within their scope, as in:

```
{
```

```
  def this : self;
  # lots of other stuff that can use "this" safely
}
```

6.6 Operator Rewrites and Operator Names

The parser and rewriter phases introduce new names into a program, either as part of transforming one form to another, or so as to protect temporary and transient values:

- The names "`0call`", "`0args`" and "`0return`" are used in rewriting functions as continuations in the rewriter.
- The name "`0noname`" is used in dealing with compile-time errors.
- The names 1, 2, 3 and the other positive integers are uses as tuple field names.
- For each defined operator, a name is created for the function into which it is rewritten. For multi-name operators, the names are joined together with a "$" between each pair. A "$" is placed at the start of the name if it has a leading argument, and a "$" at the end if it has a trailing argument. As examples, "$+$" is the function name of the "+" infix operator, and "fordo" is the name of the for-do loop operator.

These names are chosen to be easy to avoid, but also easy to use when needed.

A AFL Compiler and Run-Time Options

The AFL compiler is invoked as follows:

```
afl options inputfiles
```

For example:

```
afl /il mytest.afl
```

The AFL compiler options can be listed using:

```
afl /?
```

or

```
mono afl.exe -help
```

Experiment with options when you're not sure.

A.1 Option Syntax

All options are specified with a leading / or one or two of - (slash, dash or dash-dash). Where the option has a value the option name and value are separated by an equals sign or a colon. The following are all equivalent:

```
/k=t   /k:t   -k=t   -k:t   --k=t   --k:t
```

Non-option entries on the compiler's command line are interpreted as AFL source file names. AFL source file names that start with a / or - can be entered using a zero-length option name:

```
/=filename   /:filename   -=filename   -:filename
```

A.2 Language-Modifying Options

Not all the options are described here. Only those that make a difference to how the language works:

`/k=k` Require stropping of word-form keywords.

With this option, for example, "atEnd" is not a keyword, but is a name. To use the keyword, use "'atEnd", with a prefixing quote ("'").

`/k=m` use CR+LF for line end.

`/k=n` copy line-ends as-is. Line breaks in strings are normally interpreted as line-feeds (a.k.a. control-J). These two options change that behaviour. /k=m says interpret line breaks as carriage-return plus line-feed, the MS-DOS/MS-Windows convention. /k=n says copy line breaks as-is – as they appear in the source program. /k=n can be useful or dangerous – it's behaviour can change from one machine to another.

`/k=t` Enable template mode. In template mode, the whole of this AFL source program is treated as if it were a single string. AFL is incorporate in the program using the \{ ... } notation. /k=t illustrates that programming languages and "template" languages differ only by whether the top-level of the program starts within code or within a string.

`/k=x` Don't recognize XML-style strings. When specified, things like <a are recognized as a "<" operator followed by the letter "a". With judicious use of spaces around names, it shouldn't be necessary to ever use /k=x.

`/s=a` Add "returnFrom" to functions. Normally the "return" operator requires an outer dynamicly wrapping "returnFrom". When /s=a is specified, every function defines a "return". In the absence of a nested returnFrom, a return returns a value from the current function. Return from a "void" function using something like:

```
return nil
```

`/s=o` Use curried form for operator arguments. Normally an operator is interpreted as invoking a function with an argument list of however many arguments are defined for the operator. When /s=o is specified, an operator is interpreted as invoking a function with one argument in the non-argument-list form. If there's more than one argument, calling the function with the first argument returns a function that accepts the second argument, and so on until all arguments have been accepted, and the last function's result is returned. For example, with the definition:

```
def [150 a] + [151 b]: ...;
```

The associated function is defined as:

```
def #$+$ (a, b): ...;   # normally, without /s=o
```

or

```
def #$+$ {a} {b}: ...;   # with /s=o
```

A use of "A + B" is interpreted as either:

```
$+$ (A, B)    # normally, without /s=o
```

or

```
$+$ {A} {B}    # with /s=o specified
```

`/use=FILENAME` Normally, the AFL2 language is defined by reading the file "afldefs.afl", and compiling the AFL source as if it were in a frame following the definitions in that file. The /use option causes "afldefs.afl" to not be read in, and AFL2

to be defined by the specified file instead. Because most of the language is defined by the "/use" file, you can define your own language this way.

`/use-` specifies that no AFL2-defining file is read in, and the source program is interpreted as "raw" AFL1. This mode is useful for testing out new forms, where you don't want existing definitions to get in the way.

A.3 Run-Time Options

Run-time options are specified in the same manner as those for the compiler. What's currently available is:

`/d` Always display the value returned from the main program.

`/e` Display extra information in frame and continuation string representations.

`/tr` Enable tracing (requires that the AFL program was compiled with /tr as well).

`/uN` Set the number of "parent" levels displayed on exceptional termination to N.

`/v` Always terminate verbosely, as if with an exception.

Any unnamed command-line arguments, and all those following a stand-alone "-" on the command line are passed to the program as an `arraylist` named "`args`".

B AFL0 Syntax Specification

This is what's left language-wise after including all the **include**s and "afldefs.afl", and doing all the rewrites, prior to the "flattener" rewrite that leaves the thing no longer AFL:

```
program = frameContent

frameContent = ( ( "def" name ":" expr )*
                   "def" name ":" expr ";" )?
               expr ( "atEnd" expr )?

expr = "catch" ":" expr |
       "'{" "}" ":" expr |
       primary

primary = "{" frameContent "}" |
          "(" ( expr ";" )* expr ")" |
          "'{" ( ( expr ";" )* expr )? "}" |
          name | number |
          string ( "{" expr "}" )? |
          "self" | "throw" | "contArg" |
          "exceptionHandler" |
          primary "'{" expr "," expr "}" |
          primary "." name |
          primary "." "exceptionHandler"
```

C AFL1 Syntax Specification

This is everything minus the operators and functions defined in "afldefs.afl":

```
program = ( "externalModule" string )* frameContent

frameContent =
    ( defPart ";" )? exprPart atEndPart? ";"? |
    defPart? atEndPart? ";"?

defPart = ( exprPart ";" )? def
          ( ";" ( exprPart ";" )? def )*

atEndPart = "atEnd" exprPart?

exprPart = ( expr ";" )* expr

def = "def" opDefArg? name ( opDefArg name )*
            opDefArg? functionArgs? ":" expr

def = "def" name "[" "]" functionArgs? ":" expr

def = "def" ("[" | "'[")
                  ( name ( "," name )* )? "]"
            functionArgs? ":" expr

def = "def" name "'{" (name | "contArg")? "}"
      ":" expr

def = "def" name ( "=" expr )?

def = "def" "op" opKindArg? name
      ( opKindArg name )* opKindArg? ":" expr

def = "def" ("[" | "'[") ( "*" ( "," "*" )* )? "]"
            ":" expr

functionArgs = ( "{" name? "} |
                 "(" ( name ( "," name )* )? ")" )+

opDefArg = "[" precedence name? "()"? "]"

opKindArg = "[" precedence "*"? "()"? "]"

precedence = number |
             "(" ")" | "[" "]" | "{" "}" |
             "'(" ")" | "'[" "]" | "'{" "}"

expr = "catch" ( name | "throw" )? ":" expr

expr = functionArgs ":" expr

expr = "'{" ( name | "contArg" )? "}" ":" expr

expr' = name expr' ( name expr' )* name?

expr' = expr' ( name expr' )+ name?

expr = primary

primary = primary "." name

primary = primary "{" ( expr ( ";" expr )? )?  "}"

primary = primary "(" ( expr ( "," expr )* )? ")"

primary = primary "'{" ( expr ( ";" expr )? )?  "}"

primary = "{" frameContent "}"

primary = "(" ( expr ";" )* expr ")"

primary = "'(" ( ( expr ";" )* expr )? ")"
```

```
primary = "'{" ( ( expr ";" )* expr )? "}"

primary = name

primary = number

primary = string

primary = "self"

primary = "throw"

primary = "contArg"

primary = "exceptionHandler"

primary = "nil"
```

Unlike the other quoted tokens in this syntax, "=" in the "def" production is a name not a keyword. So it can be defined as an operator, for example.

The expr' productions are informal placeholders for the invocation of user-defined operators.

D Operator Precedence

Operator precedence determines how arguments bind to operators in the absence of parenthesization. Arguments bind more tightly or more loosely to operators. For example a + b * c is parsed as if it were entered a + (b * c) because arguments bind more tightly to * than to +.

Operator precedence in AFL is represented by a non-negative integer value. The larger the precedence number, the more tightly it binds to its adjacent operator name. For infix operators, the relative values of the right and left argument precedences determined whether an operator is "left-associative" or "right-associative": higher value on the left, left-associative, higher value on the right, right-associative.

The following lists are provided as a summary of the standard operators provided with the current release of AFL in the preamble file "afldefs.afl".

D.1 Infix Operators

```
91 = 90
91 ||= 90
91 &&= 90
91 +=  90
91 -=  90
91 *=  90
91 /=  90
91 \=  90
91 //= 90
91 \\= 90
91 ++= 90
91 **= 90
91 << 92
120 || 121
121 && 122
139 == 139
139 < 139
139 != 139
139 <= 139
139 > 139
139 >= 139
130 :+: 180
130 :-: 180
131 :++: 132
132 inner 133
132 outer 133
134 forever
134 once
134 whilst 135
136 by 137 to 136
136 by 137
136 to 136
136 map 137
136 next 137 to 136
136 next 137
140 ++ 141
141 ** 142
142 take 143
142 drop 143
150 + 151
150 - 151
150 -- 151
160 * 161
160 / 161
160 \ 161
160 // 161
160 \\ 161
181 field 182
181 hasField 182
```

D.2 Prefix Operators

```
critical 90 do 90
do 90 until 90
for 90 do 90
if 90 then 90 else 90
if 90 then 90
loop 90
print 90
printinline 90
resignal 90
return 90
returnFrom 90
signal 90
try 90 do 90
try 90 except 90
while 90 do 90
<< 92
! 122
each 136
+ 151
- 151
* 180
ccall 180
char 180
embeddedValue 180
length 180
ord 180
ref 180
repr 180
```

```
sqrt 180
toNumber 180
toLower 180
toUpper 180
typeOf 180
```

D.3 Operators With No Arguments

```
now
```

E References

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[2] D. Berry. Introduction to Oregano. In *Proc. of the ACM Symposium on Data Structures in Programming Languages*, SIGPLAN Notices, pages 171–190, February 1971.

[3] Canadian Government Printing Office, Hull, Quebec. *The HUGO Language Manual and Report*, August 1980.

[4] R. E. Griswold and M. T. Griswold. *The Icon Programming Language*. Peer-To-Peer Communications, New York, third edition, 1997.

[5] D. Megginson. The SAX Project. `http://www.saxproject.org`.

[6] Norwegian Computing Center, Oslo. *Simula Users Guide*, revised edition, 1975.

[7] The Scheme Programming Language. `http://schemers.org`.

[8] Omnimark Developers' Resources. `http://developers.omnimark.com`.

[9] D. Thomas. *Programming Ruby*. Pragmatic Bookshelf, Raleigh, second edition, 2004.

[10] Document Object Model (DOM). `http://www.w3.org/DOM`.

[11] XML Pull Parsing. `http://www.xmlpull.org`.