Beast

PROGRAMMING LANGUAGE

NEW OBJECT-ORIENTED C-FAMILY IMPERATIVE LANGUAGE
WITH VAST METAPROGRAMMING AND COMPILE-TIME EXECUTION POSSIBILITIES

BASIC LANGUAGE PROPERTIES

- Imperative
- Object-oriented (C++ class model*)
- C-family syntax
- Import-based module system* (Java, D, C#, ...)
- Multi-purpose
- Influenced by the D programming language
- Const-by-default* (suffix operator ! for mutability)
- Rebindable references (syntax is `Type?`)

DECORATORS*

Can be used in many different ways:

- Type wrappers (`@refCounted`)
- Function wrappers (`@memoize`, `@synchronized`)
- Compile-time metadata (`@description`)
- Access modification (`@public`, `@friend`)

DRAGON (PROOF-OF-CONCEPT COMPILER)

- Written in D programming language
- Multi-threaded
- Open-source, MIT licence
- Continuous integration test suite
- Transcompiles to C

THE `:IDENT` SYNTACTIC CONSTRUCT

- Uses inferred type namespace for identifier resolution instead of the current namespace (for function call arguments, the type is inferred from the parameter)

```
1  // Without :ident
2  File f = File( "src.txt",
3    File.OpenMode.read | File.OpenMode.write
4    );
5  // With :ident
6  File f = File( "src.txt", :read | :write );
```

CODE HATCHING CONCEPT - COMPILE-TIME EVALUATION, TEMPLATES, METAPROGRAMMING

- Based on variables that are evaluated at compile time (decorated with `@ctime`)
- Compile-time and standard variables can be used simultaneously in the code
- Unifies compile-time function execution, templates*, reflection* and metaprogramming in general
- Useful for code optimization
- Type variables

```
1  auto func( @ctime Type T, @ctime Int f, Int i ) {
2    // x is evaluated at compile-time
3    @ctime Int! x = factorial( f );
4    Int y = f * i;
5    x += 5;
6    print( y + x );
7
8    T result = y;
9    return result;
10 }
```

CONSTANT-VALUE PARAMETERS

```
1  class Stream {
2
3  @public:
4    Void #ctor( CreateFrom.fromFile, String filename ) { /* ... */ }
5    Void #ctor( CreateFrom.fromString, String str ) { /* ... */ }
6
7  }
8
9  Void main() {
10   Stream str1 = Stream( :fromFile, "file.txt" );
11   Stream str2 = Stream( :fromString, "asdfgh" );
12 }
```

- Template specialization replacement
- Useful for differentiating two overloads with same parameters

* THESE FEATURES ARE NOT (FULLY) IMPLEMENTED YET IN THE COMPILER
HOWEVER, THEY ARE PART OF THE LANGUAGE DESIGN

EXCEL@FIT ARTICLE

GITHUB