Introduction

Programming languages can be broadly divided into two: *general purpose* programming languages are intended to solve a wide array of problems; whereas *task specific* programming languages are more finely tuned to the solution of a narrower class of problems.

Although capable of addressing many problems, FLAS is unashamedly in the second category. It aims to solve two problems well, one on either side of the web client/server divide. On the client side, it aims to provide developers with the ability to create gorgeous, snappy, reactive applications with ease; and on the server side it assists them in building the kind of reactive microservices that underpin those client applications.

Moreover, there is almost no area of programming about which FLAS is agnostic; it has strong opinions on almost every big debate in programming. If you don't like that, then you probably want to go elsewhere. But to avoid later confusion, we would like to state up front that these are deliberate, unquestionable opinions baked into FLAS:

- **Testing at every level of scale** is an **absolute imperative** and it will be supported, encouraged and demanded of FLAS applications.
- Programs should be **reactive**, **tell-don't-ask** and should **subscribe** to event sources; they should not use getters, request/response or synchronous technology or the appearance of synchronicity.
- Logic should be **clear, transparent and provable** using functional, declarative semantics.
- The **iteration length** of any action should be as short as possible, promoting comprehensibility of code blocks.
- Programs should be **strongly typed** but with a minimum of declaration and a maximum of **inference**.
- Programs should be **broken down** appropriately supported by suitable **building blocks**.
- Programs should be as **loosely coupled** as possible, and the **language** should have all the relevant **constructs** to support that.
- **Program divisions** should, where possible, not be arbitrary but **flow naturally** from the information flow in the system.

FLAS cannot be a general purpose programming language because it makes too many assumptions about the kinds of programs - or more specifically, *program units* that you want to write. It knows about three basic program units and assumes that you are working towards one of them:

- *Cards* support the construction of UI units, combining data and screen real estate while being connected into a wider ecosystem through *contracts*.
- *Agents* facilitate the coordination of cards by combining data with a network of connections to cards and services.
- *Services* support the construction of server-side *microservices*, embedded and deployed within Ziniki[1] servers.

Supporting these three units are a number of other building blocks - *structs*, *unions*, *contracts*, *objects* and *tests* which provide the ability to build more general purpose units for program composition.

FLAS programs are intended to have semantics that are detached from any implementation language. Currently, it is possible to generate JavaScript and JVM bytecodes from FLAS, although technically there is no reason not to generate backends compatible with iOS, .NET, PHP or any other environment.

[1] Technically, since FLAS generates JVM and JS code, services could be deployed within servers provided by any PAAS provider, but for obvious reasons, we will only consider FLAS services embedded in Ziniki servers.

1 Lexical Conventions

FLAS programs are presented to the compiler as a set of *files*, grouped into *packages* (directories). The *directory name* is used as a package prefix for all the definitions provided within that directory.

For standard program units, the name of the file is not used in naming FLAS constructs, although it is used to partition test classes into different subpackages, where a variant of the file name is used as a nested package within the directory name.

1.1 Unit Translation Types.

```
(1)              file  ::=  source-file
                         |  unit-test-file
                         |  protocol-test-file
                         |  system-test-file
```

For each file within a package, the file extension is used to determine how the contents of the file will be interpreted.

- `.fl` - a standard program unit, which may contain any normal definitions.
- `.ut` - a unit test file, containing unit test definitions.
- `.pt` - a protocol test file, defining tests that can be used to test the compliance of instances to the expectation of a contract.
- `.st` - a system test file, that is capable of simulating end-to-end system behaviors and asserting their correctness.
- `.fa` - an assembly file indicating how a deployable client or server unit can be built from components in this (and potentially other) packages.

1.2 Indentation

Within a file, the nesting of definitions is determined by *indentation* and *context*. *Significant indentation* is provided using leading `tab` characters, while *continuation lines* have the same number of `tab` characters as the preceding line but additional `space` characters to reach the desired continuation point.

- By convention, FLAS programs are presented with tab stops set at four spaces, but there is no significance to this. It is hoped that tools (such as IDEs) will make the distinction between leading tabs and continuation spaces very clear and clearly present mismatched indentation as such.

Comments

Blank lines, lines beginning with no tabs, and any portion of a line following two consecutive slash characters (`//`) are considered to be *comments*. For all practical purposes, after making this determination, comments are ignored by the compiler. They may, however, be used by other tools for the purpose of providing documentation or otherwise.

- Specifically, it is the designers' intent that *literate programming* should be supported by tools. To further this, comments that begin in column zero with no leading slashes should be considered *literate comments*. These are the comments which would be used to construct documentation and narratives about the code. On the other hand, comments beginning with a double slash will generally be considered side-notes by and to developers which are to be ignored by all tools.

  Comments with meta-tags (such as `TODO`) may at some point be considered by some tools to be special, but no guidance can be given at this time in the absence of such tools.

□

Nesting

Top-level definitions are identified by having exactly one leading `tab` character. Any such definition has global visibility with its *package-qualified name* and visibility within the package with its *simple name*.

A definition may be nested within another definition provided it has exactly one more leading tab than the enclosing definition. The rules of what may be nested within a definition depend on the kind of that definition, as do the rules of name scoping; but no nested definition is *directly visible* outside the scope of its parent definition.

- The rules here are varied and complex, but fundamentally constants, functions and standalone methods are invisible outside their scope; things like fields and object methods are visible *within the context* of a container of the enclosing type; and things such as conditional definitions are visible as *part of* the enclosing definition.

  One implication of the "one more leading tab" rule is that each line in a FLAS program must have less, the same or exactly one more leading tab than the preceding line (ignoring comment lines).

□

1.3 Names

FLAS supports four types of names with different lexical rules:

- **field names** must start with a lowercase letter, followed by an arbitrary number of lowercase and uppercase letters, digits and underscores. CamelCase is used to indicate word boundaries.
- **type names** must start with an uppercase letter, be at least three characters long, and have the remaining characters be uppercase and lowercase letters, digits and underscores. CamelCase is used to indicate word boundaries.
- **polymorphic type variables** must be one or two characters long, the first of which must be an uppercase letter and the second may be an uppercase letter or a digit.
- **template names** must start with a lowercase letter, followed by an arbitrary number of lowercase and uppercase letters, digits, hyphens and underscores. While uppercase letters and underscores are permitted, lowercase letters and hyphens are generally preferred. Hyphens are used to indicate word boundaries.

These kinds of names are referenced as appropriate within this manual.

1.4 Constants

Apart from named type constants, FLAS supports constants of the builtin primitives:

- **String** constants are indicated by the use of single or double quotation marks. These may be used interchangeably to indicate the start of a string, but they **must** be used in matched pairs: that is, a string beginning with a single quotation mark continues until a closing single quotation mark is encountered; and likewise with double quotation marks. A terminating quotation mark may be embedded within a string by placing two consecutive marks in the string; one will be maintained and the string will continue.
- **Integer** constants are indicated by a sequence of one or more digits. Negative numbers are not supported as constants but rather as expressions using the unary minus operator and a positive integer constant.
- **Floating point** constants are indicated by a sequence of zero or more digits, followed by a dot (`.`), followed by zero or more digits, optionally followed by an exponent symbol (`e`, `e-`, `E` or `E-`) and one or more digits. One of the two mantissa portions must have at least one digit.

Both integer and floating point constants are considered to be of type Number.

- FLAS tries to ride two horses with regards to numbers. On the one hand, it prefers to assume (as does JavaScript) that there is just one number line and there is nothing really special about integers; on the other hand, it has to recognize that many applications (such as array indexing) *require* integers and it is not reasonable to just ignore their existence.

Integers are not formally defined in FLAS: the only recognized number type is `Number` which roughly equates to the set of real numbers. However, the implementation - particularly in the JVM world - is riddled with concern for the distinction between floating point numbers and integers.

- It is the designers' intent to directly support more primitive types, including monetary values, dates and intervals. These will ultimately have the appropriate constant types.

  Complex numbers may also be supported as primitives if the need arises, but as was noted in the introduction, FLAS is not intended to be a general purpose programming language and complex numbers seem outside the intended applications of the language.

1.5 White Space

The issues regarding leading white space have already been addressed. This section only relates to white space found *after* the first non-white-space character and *not* in a comment.

Within a line, any white space *not* occurring within a string constant is considered to end the current token and introduce a new one. Within a line, multiple consecutive white space characters are considered equivalent to a single space. Within a line, all white space characters are considered equivalent.

When a line is continued by starting a new line which begins with the same number of tabs as the previous line and one or more non-tab space characters, the compiler internally joins the lines together, removing any newline (`CR` and LF) characters but preserving any other white space (tabs and spaces) originally present.

An arbitrary number of continuation lines may be joined together in this way, all of which **must** have the same number of leading tabs; the first of which must have **no** subsequent white space; and all the others must have at least one space after the leading tabs. There is no rule about the relative number of spaces following the leading tabs - that is left to the developer's sense of clarity and aesthetics.

1.6 Punctuation Characters

In addition to names and constants, FLAS defines operators and punctuation characters.

Punctuation characters stand alone and constitute a single symbol by themselves and may **not** be combined into larger symbol characters.

The following characters are punctuation characters

- Parentheses `(` and `)`
- Brackets `[` and `]`
- Curly brackets `{` and `}`

- Comma ,

1.7 Symbols and Operators

The remaining characters are considered symbol characters and may be combined into composite *operators*. An operator is either an individual symbol character or a sequence of symbol characters which have been defined to have meaning as an operator. Some symbols may also be used in language constructs.

Currently there is no mechanism by which users can introduce new operators.

Operators may be used in expressions as well as in other contexts. They are defined in the appropriate sections where they are used, along with their precedence (where appropriate). Where symbol characters need to be placed in distinct operators which are adjacent to each other, intervening whitespace (or parentheses) must be used as appropriate.

- In the long run, it makes sense to allow new, user-defined operators. These are essentially functions which can be given arity, precedence and associativity. However, doing so in a truly extensible way (and supporting concepts such as ternary operators) is beyond the current scope. Consequently, the set of operators is currently limited to the builtin operators.
-

2 Declarations and Scopes

Within FLAS, many different types of concept can be defined; furthermore, the language is intended to be extensible so as to support additional concepts, in particular to support concepts internal to Ziniki. Each of these concepts has its own declaration syntax and then supports specific nested content. The details of these declaration types will constitute much of this manual.

However, these definitions can be broken down into "families" and these families will be discussed here.

Most definitions and nested declarations are introduced by a keyword or key operator; the exceptions are:

- when only one kind of declaration is allowed;
- function definitions.

- FLAS has been designed from the outset to be a "family" of languages. The core of the language is the ability to express functional transformations from state to state in conformance with the actor model. Anywhere that this model is applicable represents a potential target domain for FLAS.

  In this regard, concepts like "cards" and "services" make sense in some contexts and not in others; more than that, in embedding FLAS in the Ziniki context, it is desirable to have more direct support for defining new concept types such as storable *entities* with unique identifiers; *offers* and *deals* between parties and so on. These are not part of the "core" language.

  Similarly, other embedded uses offer other extensions to the core model, but in all cases the fundamental design of the language remains the same.

  -

```
(6)      top-level-unit  ::=   top-level-definition
                           |    function-scope-unit
(7) top-level-definition  ::=   struct-declaration
                           |    union-declaration
                           |    envelope-declaration
                           |    wraps-declaration
                           |    contract-declaration
                           |    object-declaration
                           |    service-declaration
                           |    agent-declaration
                           |    card-declaration
(8)      function-scope  ::=   function-scope-unit*
(9) function-scope-unit  ::=   function-case-definition
                           |    tuple-definition
                           |    standalone-method-definition
                           |    handler-definition
```

2.1 Functions

At heart, FLAS is a *functional* language, and functions are core to data manipulation in FLAS. As with most modern functional languages, FLAS functions are mathematically defined mappings from domains of values to a range of values. While not perfectly mathematical, the use of lazy evaluation ensures that the operational semantics are close to the declared semantics.

The family of function declarations should be understood to include standalone and object *methods* as well as event handlers and data callback handlers.

Depending on how the function is defined, the immediate nested members of this family may be conditional cases, which in turn introduces a scope where functions may be defined. For simple functions, defined on one line, the immediate nesting level allows functions to be defined.

2.2 Data Types

The family of data type declarations includes `struct`, `union` and `object` definitions in core FLAS, as well as being subject to extension; Ziniki also defines `entity`, `deal` and `offer`.

For these types, the immediate nesting level defines fields. Inner nesting levels allow constraints and metadata to be applied to the individual fields.

2.3 Contracts

Contracts define a set of parallel methods to be implemented by an implementing actor or handler.

Nested declarations are all declarations of methods to be implemented. Nested definitions are not allowed.

2.4 Actors

The family of actors includes agents and cards on the client side and services on the server side in core FLAS; it may also include other actors as appropriate in extension contexts.

Nested lines may be acceptable nested definitions; nested declarations must be introduced with a suitable keyword such as `state`, `template` or `implements`.

2.5 Templates

WIthin templates, no nested definitions are allowed. The nested declarations must all be binding, styling or event handling declarations.

2.6 Scoping

Declarations at the top level of a file (with exactly one leading tab) are prefixed with the name of the enclosing package (the directory in which the file is found), unless the file is a test file of some kind, in which it is prefixed by the name of the enclosing package followed by some version of the test file name (usually including invalid characters to ensure its uniqueness).

- Unit test files (`.ut` files) use the package _ut_*name*. Since the test cases themselves do not have names (just descriptions), the individual tests are given the names *ut0,* ut1 and so on. Placing the tests within separate packages (based on the file name) ensures the uniqueness of the overall name. Using underscores in the package and test names ensures that they cannot clash with names defined with FLAS, since these cannot contain underscores.

Nested lines have a meaning that is imposed by the enclosing scope.

In many cases, a nested line is constrained to provide a part of the definition of the enclosing scope. For instance, inside a function definition, a line beginning with the condition operator (`|`) forms a function case, an inherent part of the function.

However, when this is not the case, one or more definition types may be allowed; in most cases constant and function declarations are allowed wherever there is an inner scope. In this case, the name of the inner definition is prefixed not only by the package prefix but also by the name of the enclosing definition.

Within a given scope, a single name may not be used twice. Specifically, a name introduced in a scope at a given level must be unique and must not be defined in any enclosing definition. While this most obviously applies to function, type and actor names, it also applies to other names introduced into the scope such as parameter, field and type variable names.

- I feel this needs a few examples to make it absolutely crystal clear.

Some declarations - such as actors and data types - may not be nested but must be placed at the top level.

3 Environment, Start and Finish

In most programming languages, the start and finish conditions are very clear. Because of the actor model in FLAS, this is very much less clear and needs to be considered both in terms of the overall environment and at the individual actor level.

3.1 Environment Start

Browser

App Clients

Embedded Ziniki Server

3.2 Actor Lifecycle

Actor Start

Actor Termination

4 Structs and Unions

FLAS offers two data types as standard[1].

4.1 Struct Definitions

```
(10) struct-declaration  ::=   STRUCT type-name poly-var*
                          >>   struct-field-list
(14)  struct-field-list  ::=   struct-field-decl*
(15)  struct-field-decl  ::=   type-reference var-name struct-initializer? EOL
(17)  union-declaration  ::=   UNION type-name
                         >>>   type-reference
```

[1] More are defined in the Ziniki extensions, see some reference.