Templates

Templates bridge the gap between data and interaction, providing a means of combining visual display elements with `card` or `object` data.

Template Definitions

Each card may have one or more template definitions.

```
(90)  named-template-definition
 ::=    TEMPLATE template-name
   (91)  define-template-chain
 ::=    SEND
        template-chain-var+
        (92)  template-chain-var
 ::=    ORB type-name var-name
        CRB
```

Each template definition must have a corresponding visual element defined.

- The exact nature of the visual element definitions depends on the system being targeted. The default is HTML and the visual elements are defined in standard HTML files annotated with element `id`s indicating their role in the system.

  These notations are described in appendix§13.

The first template definition describes the visual appearance of the entire card. All subsequent template definitions describe the visual appearance of a sub-element of the card.

The first definition may not be referenced by any templates.

All other definitions must be referenced by a template before they are used. These may also be referenced (mutually recursively) by subsequent definitions.

A template definition consists of a set of bindings indicating how the data is to be used to prepare the visual elements.

Template Fields

Abstractly, the visual design is reduced to a set of template and field definitions. There are six kinds of definitions: each definition must have a name and be of one of these kinds. The visual design mechanism must have a way of indicating the kind and name.

Templates

Card Templates are blocks of visual design containing the entire layout for a card.

Item Templates are blocks of visual design containing the layout for a compound element within a card.

Containers

A **Container** is a block of visual design which is capable of holding zero or more items.

A **Punnet** is a block of visual design which is capable of holding zero or more nested cards.

Content

A **Content** definition is a block whose content is set and styled by the template mapping. It may also have event handlers attached.

A **Style** definition is a block whose content is determined by the visual designer but whose styling may be determined by the template mapping and to which event handlers may be attached.

Template Bindings

A template binding indicates how the appropriate block of the template is to be populated.

```
(93)  template-bind      ::=    template-name SEND
                                expression
                                pass-to-template?
                        |       template-name
                        |       template-name
                                (94)  option-template-binds
                        ::=     option-template-bind+
                                default-option-template-bind
                                ?
                        |
                                default-option-template-bind
                                (95)  option-template-bind
                        ::=     GUARD expression SEND
                                expression
                                pass-to-template? EOL
                        (96)  default-option-template-bind
                        ::=     SEND expression
                                pass-to-template? EOL
(97)  pass-to-template  ::=     SENDTO template-name
```

Bindings are defined by specifying the name of a template field as a destination and a value to use to configure that field.

A template name may not be used as the field in a binding.

A style field may not be used for binding.

A value bound to a content field must be of type `String`.

A value bound to a punnet field must be of type `Crobag`.

Template Styling

Styling may be applied to content and style fields only.

```
                          (98)  template-customization
                        ::=     template-style
                        |       template-event
(99)  template-style    ::=     GUARD expression?
                                SENDTO STRING+
```

Styles may be applied conditionally or unconditionally.

Conditional styles may be nested. A nested conditional style is only applied if all the nesting conditions are true.

- This mechanism does not provide any additional functionality than would be present without it. However, the ability to nest conditional styles makes it easier to understand the relations between styles and removes duplicated conditions.

A conditional style consists of a condition and a set of styles. The condition must be of type `Boolean`. If it is `True`, the styles will be applied and nested blocks considered; otherwise they are all ignored.

Styles must be of type `String` or `List[String]`. Constants and variables may both be used.

All of the matching and unconditional styles will be reduced to a single set of styles that will be applied.

Template Events

Event handlers may be attached to content and style elements.

```
(100) template-event    ::=     SENDTO var-name
```

The specification of a handler is the `SENDTO` operator (`=>`) followed by the name of the event handler.

Event handlers may be included within conditional styling blocks. In this case, the event handler is only registered on the appropriate element if the conditional style is applied.

Commentary

MVC

FLAS uses a variant on the MVC (**M**odel, **V**iew, **C**ontroller) pattern. In this pattern, there is a unique source of truth about data values (the Model), which is the state of the actor. The templates determine what is shown to the user (the View) by taking the provided visual elements (usually HTML) and combining them according to the instructions in the template and the data in the model. The operations supported by the actor (contracts and events) constitute the Controller.

The main intent of this design is to present a separation of concerns. In particular, by having very loose coupling between the visual elements and the data (a binding, mainly based on kind and name of slot), it is easy to separate the process of visual design from the process of coding. This is *not* an attempt to introduce silos, but the reality is that these processes *are* distinct and have distinct tools.

Development Cycle

To facilitate close working of cross-functional teams, FLAS works hard to make it very easy to reliably, repeatedly and automatically combine the output of visual design with the ongoing code development.

The latest version of the visual design is always processed and analyzed along with the code in every build (including from IDEs). The visual design may either be presented as a directory or as a `ZIP` file.

Commands are offered to directly import from visual design tools. In particular, it is possible to download the latest version of the visual design from `webflow` at the touch of a button inside VSCode.

A Mental Model

The reference given in this chapter is intentionally concise and abstract. While formally correct, it may be difficult to comprehend and apply.

The should be the starting point for developers wishing to *learn* how to use templates; however, for reference purposes, this commentary will offer a concrete understanding of how the templates are used in practice by reference to the "standard model" of using templates: HTML and CSS in the web browser.

Templates are automatically extracted from input HTML using a `splitter` which is described in an appendix (§13). Each such template is then provided in the HTML delivered to the client as an HTML `template` object in the `head` section.

In theory at least, such a template may be arbitrarily complex. Certainly, it may have multiple levels of nesting. However, the splitting process automatically removes some of the nested elements (in particular, elements used within `content`, `container`, `punnet` and also any elements used to define nested `templates`). The splitting process identifies these elements using their `id`, removes this from all elements and then annotates the nodes with `data-flas-` properties so that they can be located at runtime in JavaScript using query selectors.

This means that within the context of a template, each of the nested elements (except nested templates) identified by a compliant `id` is addressable through the template mechanism.

The template mechanism in FLAS is not there to **define** templates but to **bind** the runtime values in the state of a card or object to the compatible elements (abstractly referred to as *fields*) in the template defined in HTML.

In situations where HTML is not being used but rather some other visualization mechanism, the principle remains the same but the technology will be different. Consult the appropriate documentation for your environment.

The template definitions extracted from the visual design are not specifically attached to, or scoped within, cards in the program. Rather, a binding is made between each card and the appropriate templates. This binding is, more or less, many-to-many: each template in the visual design may be used by many cards; and each card may use many templates.

Intelligent Redisplay

While the main focus of this feature is the separation of concerns, it is also important to provide efficient and simple redisplay.

This template mechanism provides the opportunity to move all of the complex redisplay code from the user and place it in the system runtime environment.

Wherever possible, the system optimizes the redisplay of items so as to make the minimal number of changes to the display. This is beneficial both for efficiency and for the user's visual experience.

Theoretically, redisplay occurs between each full message cycle to the card; however, for efficiency reasons, multiple message cycles may be processed before the display is updated. However, it is to be expected that the display (and in particular, event handling) will not be allowed to lag noticeably.

Card Templates

Templates are, in reality, a property of *cards*. While objects support templates, and it is possible to bind structs, entities and unions into appropriate template fields, in every case this has to happen within the context of an overall card.

To put it the other way, cards are a combination of data, processing, message passing and rendering. The rendering is handled by applying the card template to the current state of the card.

There can only be **one** card template for a given card. This must be identified by it having the appropriate `id` in the visual design (i.e. `flas-card-`*name*) and by it being the first template listed in the card. The card may then define as many other supporting templates as it wants, but each of these must be referred to at least once before it is defined.

The only variables that can be referenced within a card template are the members of the state. Functions and constants at the top scope of the card or in the global state may also be used.

Content Bindings

The simplest and most common form of binding is the content binding. This allows a `String` value in the card to be literally placed in a specific template field. The value will automatically undergo any "entitifcation"[1] or other encoding process required by the visual design.

A content binding is expressed using the `SEND` operator (`<-`) indicating that the *value* on the right is to be sent to the *field* on the left. Possibly the simplest such usage would be:

```
|card Simple
||template show
```

[1] Entitification is the process by which HTML replaces special characters with the appropriate HTML *entities* to stop them being interpreted as HTML. It is not possible to use FLAS to inject arbitrary HTML into a template. This would be both a security and a portability issue.


```
|||hello <- "hello, world"
```

This places the constant `String` value `"hello world"` in the template identified by `flas-card-show` in the field identified by `flas-content-hello`.

Because all content fields require `String` values, it is necessary that non-`String` values be converted to `String`s before binding. This does not happen by default, but the `show` function may be used to do the "default" conversion.

```
|card Simple
||template show
|||quantity <- show 15
```

This is obviously not ideal in all cases: dates, for example, will generally want more tailored conversion (the default for `Instant` is to show the number of days that have elapsed since the beginning of 1970). In these cases, it is the developer's job to consider the appropriate transformation and use it.

```
|card Simple
||state
|||Calendar cal
||...
||template show
|||now <- cal.showIsoDate (Instant.now)
```

The process of initializing the calendar object using the `Lifecycle` contract has been elided.

Each field may only be bound once during a template definition.

Conditional Bindings

It is also possible to use guarded bindings, analogously to using guarded equations in function definitions.

In this case, the binding only contains the name of the field to be bound, and then a nested block includes the binding cases.

For example, it may be desirable to display a specific message when there are no entries, but otherwise to display a count.

```
|card Counter
||state
|||List[String] entries
||template showCounts
|||count
|||| | length entries == 0 <- "no entries"
|||| | <- (show (length entries)) ++ "entries"
```

As seen here, it is possible to end the binding definition with a single, optional default binding which applies if none of the preceding guards evaluate to `True`.

Styling

Both content and style fields may have styling attached to them. In HTML, this amounts to allowing the `class` field on an element to be set to a set of values. These values are determined by considering all of the possible styling bindings and building a composite list.

All style applications must be introduced in a nested (indented) scope.

- Style applications which are indented directly from the template are applied to the template as a whole (that is, they are applied to the HTML element associated with the template itself).

- Style applications which are indented from a `content` or `style` binding are applied to the corresponding element in the template.
- Style applications which are indented from a previous style application are applied to the same element as the enclosing application and *only if* the enclosing application is also applied.

All values on the right hand side of a styling binding must be of type `String` or `List[String]`.

All styling applications are introduced using the `GUARD` operator (`|`) and the styles follow a `SENDTO` operator (`=>`).

So, starting from the simplest case again, it is possible to apply the `bold` style to an entire template unconditionally.

```
card Styling
|template message
|| | => "bold"
```

On a content field, it is written as a nested block within the binding:

```
card Styling
|template message
||message <- "hello"
||| | => "bold"
```

Although it starts to look a little scrappy, it can also be applied to a conditional nested block:

```
card Styling
|template message
||state
|||Boolean sayHello
||message
||| | sayHello <- "hello, world"
|||| | => "bold"
||| | <- "goodbye"
```

```
|||| | => "italic"
```

- This is, in fact, why the conditional binding syntax exists. Without it, it would be simple enough to define a binding to a function which used guarded equations to deduce the desired value. But the logic to style each case separately would be both tortuous and duplicative.

It is possible to do the same thing on a styling field:

```
card Styling
|template message
||styleMe
||| | => "bold"
```

Note that because a styling field does not have a value to bind, only the name of the field is written, and no `SEND` operator is used.

Conditional Styling

In the same way in which bindings can be applied conditionally, styles may be applied conditionally. The syntax is essentially unchanged: a boolean expression is simply inserted in between the `GUARD` and `SENDTO` operators. This expression must be identifiably of type `Boolean` and, like the other expressions in the template code may only depend on state members along with global constants and functions.

```
card Styling
|state
||Boolean wantBold
|template message
||styleMe
||| | wantBold => "bold"
```


Note that, unlike guarded equations and content bindings, *all* of the cases which match will have their associated styles applied. Because of this, the order in which the applications are presented is irrelevant.

It is again possible to provide default style applications which will be applied in all cases. Again, unlike guarded equations and conditional bindings, it is possible to present any number of default style applications and all of them will be applied.

Note that there is no specific syntax for "either this or that". This is, of course, possible to achieve by using both a condition and its negation:

```
card Styling
|state
||Boolean wantBold
|template message
||styleMe
||| | wantBold => "bold"
||| | !wantBold => "quiet"
```

Nested Conditionals

A style application may contain blocks of style applications. This has two purposes.

Firstly, it allows applications that are "related" to be written closely together but on separate lines, making the intent of the code clearer.

Secondly, because the nested applications are only applied if the parent application is applied, it makes it possible to unravel complex conditionals into a block.

For example, it is possible to have a button which can be enabled or disabled and, if enabled, can be highlighted to indicate that it has notifications.

```
card Button
|state
```

||Boolean isActive

||Boolean hasNotifications

|template message

||button

||| | isActive => "enabled"

|||| | hasNotifications => "alert"

Event Affordance

Event handlers are defined in code on the card or the template. By default, when defined, they are attached to the *whole card*. While this is a useful default, it does have the bizarre property of meaning that attaching them to fields in the template reduces their area of applicability. However, in general, event handlers will either be intended to operate at the card level and will not be attached; or will only be relevant to certain nodes and will always be attached.

The act of attaching the name of an event handler to a content or style field is called an *affordance* because it makes it possible to invoke the handler from that location. Note that *no extra information* is required - or can be provided - because the event handler already contains the information about the event that it will handle, and an event handler is invoked as a message, so only two pieces of information are in scope - the current state of the card and the event object itself[2].

The event affordance is placed in a nested block of either a content binding or a style application. If it is applied to a conditional binding or conditional style definition, the event is only enabled if the containing guards are true.

Adding the affordance simply consists of using the `SENDTO` (`=>`) operator followed by the name of the event handler.

card Event

|template clickHere

[2] Although note that, because the event object contains a pointer to the value being bound, it is possible to customize the handling of the event based on which affordance was used.

||handleMe

|||=> updateMe

|event updateMe (ClickEvent ev)

This event only adds the handler if the button is active:

card Button

|state

||Boolean isActive

||Boolean hasNotifications

|template message

||button

||| | isActive => "enabled"

|||| => updateMe

|||| | hasNotifications => "alert"

|event updateMe (ClickEvent ev)

Object Templates

Object templates may be defined in exactly the same way that cards do. However, objects may not use templates specifically identified for cards but most only bind to *item* templates.

An object definition may define multiple templates, each of which may reference any of the others. There are no ordering constraints on object templates. Object templates do not need to be used.

If an object contains other objects, an object template may bind to a rendering of one of the nested object's templates.

Containers

Containers serve four overlapping roles:

- They allow a level of abstraction and encapsulation by enabling a separate template to be used to render a (structured) part of the card's data.
- Extending this, they allow `Objects` to be responsible for rendering their state into a particular part of the window.
- They allow `Lists` and `Crobags` to render multiple items - possibly of different types - into a visual list.
- By using *punnets* it becomes possible to delegate more completely to other (explicitly or implicitly typed) cards.

Punnets will be considered in the next section. The remainder of this section simply deals with *item containers*.

To support containers, additional item templates need to be provided in the visual design. These must then be bound in the card using a `template` definition.

The simplest case is the delegation of the rendering of a `struct` into a container using a template.

```
struct Combo
|String name
|Number count
card Delegated
|template main
||entry <- (Combo "ducks" 22) => combo
|template combo
||item-name <- name
||quantity <- count
```


Here, the `main` template (which must be a card template) has a defined container called `entry`. It is bound to the constant value `Combo "ducks" 22` and this value is formatted using the `combo` template. Because of this specific use of the template name, the template knows (through type inference) that it is being applied to a `struct` of type `Combo`, and thus the names `name` and `count` are in scope - referring to the fields of the `struct`.

This template can be referenced multiple times within the card, but on every occasion it must be used to render a `struct` of the same type.

Delegating to object templates works in much the same way, except that the template is nested inside the object rather than in the parent card.

```
object Combo
|state
||String name
||Number count
|ctor make (String name) (Number cnt)
||...
|template combo
||item-name <- name
||quantity <- count
card WithObject
|state
||Combo thing
|...
|||thing <- Combo.make "ducks" 22
|template main
||entry <- thing => combo
```

Here, the `object` has to be created in a message-method context, the details of which have been elided.

The `combo` template is on the object, as is the state which it references. In this case, because the delegation is through the object, no additional variables are introduced - the references are to members of the object's state.

Objects may have any number of templates, but they must all be *item* templates.

1 Introduction

Programming languages can be broadly divided into two categories: *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.

2 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.

2.1 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.

2.2 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 initial 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 can be referenced from any scope using its fully qualified *package name* and from within the package using its *simple name*.

A definition may be nested within another definition provided it has exactly one more leading tab than the enclosing definition. The grammar defines which constructs may be nested in which context.

Some definitions (such as `structs`) may include sub-definitions in indented lines which are accessible in expressions which have first identified the enclosing definition.

Otherwise, nested definitions are not visible from outside the scope of the enclosing 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.

□

■

A further 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).

□

2.3 Names

FLAS supports four types of names with different lexical rules:

- **variable names** and **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.

2.4 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.
- **Numeric** 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 frequently resorts to testing whether a number is an integer before carrying out integer-only operations.

□

■

There are more builtin, primitive types (such as `Instant`, `Interval`, `Currency` and `Money`). However, these do not have any associated constants but must be constructed using the appropriate functions.

□

2.5 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.

2.6 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 `,`

2.7 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.

3 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 an overview of 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
                        |      entity-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
```

3.1 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.

3.2 Data Types

The family of data type declarations includes `struct`, `union` and `object` definitions in core FLAS; Ziniki also defines `entity`, `envelope`, `deal` and `offer`, along with the entity mapper declaration `wraps`. The `state` of `objects` and `cards` can be considered very similar.

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

3.3 Contracts

Contracts define an abstract mechanism by which interactions between cards and services can be defined (similar to interfaces or protocols in other programming languages). Because they are simple declarations, they are generally very simple.

One level of nesting is permitted to contracts to define the individual methods supported by the contract.

3.4 Actors

The family of actors includes `agents` and `cards` on the client side and `services` within microservice providers.

The top level of nesting within these definitions describes the individual elements that make up the overall definition. These inner definitions are defined in the grammar.

Arbitrary function and standalone methods may also be included at the top level of nesting.Each of these is defined using a series of blockNested lines may be acceptable nested definitions; nested declarations must be introduced with a suitable keyword such as `state`, `template` or `implements`.

3.5 Scoped Names

In order to ensure that names are globally unique, the *simple* name of any definition is prefixed with the current scope name.

At the top level, the scope name is the package name.

- The source of test files (`.ut`, `.pt` and `.st` files) are collocated with the main (`.fl`) files. However, they do not have individual names and instead are placed into test-specific sub-packages of the main package, each of which is given a name derived from the file name (e.g. _ut_*file*). Since the test cases themselves do not have names (just descriptions), the functions generated for these 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. These names are never accessed externally but only by the test runner.

Nested definitions use the enclosing definition name as the scope name.

- So, for example, if there is a simple `struct` definition in package `test.my`:

  struct Thing

  String name

  the `struct` will be given the qualified name `test.my.Thing` and the field `name` will have the qualified name `test.my.Thing.name`.

The simple name introduced in a scope must be unique and must not be defined in any enclosing definition.Function definitions may, however, be defined in multiple clauses at the same level provided such definitions are consecutive; this does not violate this rule because these clauses are combined to form a single 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.

  For example, in the following definition, the parameter `x` conflicts with the enclosed definition of `x`, creating an ambiguity and is therefore disallowed:

  f x = y

  y = 2 * x

  x = 14

  In the definition of `y`, which use of `x` was intended?

4 Lifecycle

In most programming languages, there is a concept of an application and some amount of code which is considered to be "initialization" code, such as a `main()` method.

In FLAS, there are two components which have a lifecycle: `cards` and `agents`. Other component instances which may be included within these are initialized by them. In the code, no card is marked in any way as being responsible for application initialization.

- For simplicity, consider a `card`. It may be embedded within another card, or it may be started at the top level. Most cards will have an expectation about which they are going to be, but in the end, they simply can't tell.

  The default runtime container considers the configuration defined in a `.fa` file and looks for the main card. This is the one which it will start at the top level.

  However, it is also perfectly reasonable to imagine a situation in which another website links to an environment which sets up a card and runs that as if it were at the top level. Providing all the services are put in place, it is unable to tell the difference.

In Ziniki, `services` may be considered to have a lifecycle, but because they do not have state, they also do not implement the `Lifecycle` contract.

Because `services` represent multi-threaded microservice objects, they cannot have independent state and therefore do not need initialization *per se*. They do, of course, need to be instantiated and connected to other services using the `requires` construct but this is handled invisibly.

4.1 Wiring up

When a `card` or `agent` is first created, its storage is created and any state initializers are evaluated.

Any services it requests through `requires` directives are located and handles provided.

Services which cannot be found are initialized to appropriate `NoSuchService` services.

A `NoSuchService` service handles all of the requests for the service but never responds. It does, however, report on the system log that an appropriate service could not be found. Depending on the environment, the system log may not be visible.

It should be possible to test if a service has connected successfully and to react accordingly.

4.2 Lifecycle contract

`cards` and `agent` may choose to implement the `Lifecycle` contract.

The `Lifecycle` has four initialization methods and one termination method, which are called by the container synchronously after the card has been wired up but before it can start processing any requests.

`Lifecycle.init`

The `init` method is called immediately after the services have been wired up.

The result of the `init` method is fully processed before any more methods are called.

`Lifecycle.load`

The `load` method may or may not be called. The `load` method is called if there is data to present to the card. If the method is called, it will be passed an `entity` which will be the root entity which the card is expected to render.

The result of the `load` method is fully processed before any more methods are called.

`Lifecycle.state`

Cards may choose to store state *about* the rendering of an object, for example, to remember the options associated with user controls on the card which it is not appropriate to store in the entity itself. If an entity has been loaded using the `load` method **and** this card has previously recorded state about this entity, then the `state` method will be called after the `load` method with an entity representing the stored state of the object.

The result of the `state` method is fully processed before any more methods are called.

`Lifecycle.ready`

Once the result of the `state` method has been fully processed, the card will be ready to respond to user messages.

In general, an actor will arrange its lifecycle in such a way that the `ready` method will be responsible for most of its configuration. The `init` method is only generally used to do default configuration which might be overruled by later considerations in the `data` or `state` methods.

□

`Lifecycle.closing`

When a card or agent is being disposed, the `closing` method will be called to allow the card to take any last minute actions. These actions **may not** interact with the user.

■

The subject of resource management is discussed elsewhere, but in general cards and agents are disposed when one of two situations occurs:

- the browser (or browser window) is closed and the entire application is going away, or equivalent actions in other environments;
- the card is part of a punnet in a parent window that is being emptied or closed.

In particular, cards expect to be nested and punnets will hold nested cards. When the user selects a different view, or different entity to view, the current set of nested cards will be closed and new ones opened.

□

Commentary

4.1c Containing Environments

FLAS is agnostic as to how the containing environments initialize the system and create the cards.

The only consideration from a FLAS perspective is when the `card`, `agent` or `service` comes into being, it has been correctly connected (abstractly) to the external services it requested. It is likewise agnostic with regards to whether those services are local or remote. Indeed, much of the power and simplicity of the FLAS model comes from this very ignorance of the wider world.

But it is worth spending at least a moment considering the different environments in which FLAS can be deployed.

Browser

The most common deployment for FLAS is the ubiquitous browser. By providing a suitable HTML wrapper, any `card` or `agent` can be deployed in a browser running JavaScript, either as the main window or within an iframe. It is also possible to load the code for individual cards into another web application.

Phone Apps

FLAS has been designed with portability and efficiency in mind. It is possible to generate JVM bytecodes from FLAS and then assemble an Android app using an appropriate container library. This allows combinations of cards to be deployed as applications and individual cards to be deployed in a manner which is interoperable with these applications.

The same approach could be used for Apple iPhones, but there are issues with clearing the hurdles to provide such applications through the AppStore.

Microservice Containers

Although `card`s and `agent`s cannot be deployed into microservice containers for performance and threading reasons, it is possible to deploy `service`s in this way.

The container would obviously need to support the configuration of `service`s and provide the appropriate downstream services on which the `service` object depends.

Ziniki is the recommended microservice container for this purpose.

Embedded in Applications

Because FLAS offers a code abstraction, it is possible to use it in other applications if they have been designed to interact through contracts.

Typically, such applications will prefer to interact with `agent`s, but there are use cases where the template support of `card`s might be desirable.

Ziggrid, Modeller and WebPresenter are examples of applications that can embed `card`s and `agent`s.

5 Expressions

Expressions form the building blocks of FLAS programs.

```
(48)  expression        ::=   literal
                        |     var-name expression*
                        |     UNOP expression
                        |     expression BINOP
                              expression
                        |     expression COLON
                              expression
                        |     ORB expression CRB
```

FLAS is a strictly typed language with type inference. Every expression has an associated type.

5.1 Literals

The simplest expressions are literal values.

```
(49)  literal           ::=   NUMBER
                        |     STRING
                        |     TRUE
                        |     FALSE
                        |     list-literal
                        |     object-literal
(50)  list-literal      ::=   OSB CSB
                        |     OSB expression
                              comma-expression* CSB
(52)  object-literal    ::=   OCB CCB
                        |     OCB object-member
                              comma-object-member*
                              CCB
(53)  object-member     ::=   object-key COLON
                              expression
(54)  object-key        ::=   var-name
                        |     STRING
```

```
                (55)   comma-object-member
        ::=      COMMA object-member
```

Numeric Literals

Numbers are expressed according to the regular expression `[0-9.e+-]+`.

All numeric literals are non-negative. Negative numbers are expressed using the unary negation operator applied to a numeric literal.

All numeric literals are of the primitive type `Number`. The FLAS type system does not distinguish between integers and floating point numbers, although runtime checks are performed when integers are required.

String Literals

Strings are quoted and must have balanced quotes on a single line. Literals that need to be broken can be placed on separate lines and concatenated using the ++ operator.

Either single (') or double (") quotes may be used to define strings, but they may not be mixed.

A duplicated quotation mark within the string allows the quotation mark to be included in the string.

String literals are of type `String`.

- A string literal such as

  'hello ''world'''

  will evaluate to

  hello 'world'

  although of course it is much clearer to write

  "hello 'world'"

List Literals

Lists may be defined directly by writing expressions within square brackets (`[...]`) and separating items with commas (`,`).

The empty list literal (`[]`) is of type `Nil`; all other list literals are of the type `Cons[T]` where `T` is a polymorphic type variable most accurately describing the types of the members. If no better type can be determined, `Any` will be used for the value of `T`.

Tuples

A tuple is a group of two or more values whose types may be unrelated.

They are typed as `TupleN[A,B,C…]` where `N` is the number of values in the tuple and `A`, `B`, `C`... are the types of the tuple values.

Hash Literals

Hash literals may be specified using JSON-like notation.

The keys may be string literals or variable names. If variable names are used, they will be converted into the corresponding string literal as if they had been placed in quotes.

All hash literals are of the type `Hash`.

- The use of variable names as keys is a convenience as provided in JavaScript. However, the range of names is not limited to those which can be expressed as FLAS variable names, so the range of possible keys using string literals is greater than that of variable names.

5.2 Function Calls

Functions expressions are written as the name of the function followed by zero or more arguments. No additional syntax is required.

Function argument binding has the highest precedence in FLAS, so when it is desired to have an expression be an argument, it must be placed in parentheses.

A function of no arguments is a constant.

All function definitions must be strongly typed, although it is possible to define mutually recursive functions whose types are inferred together. The type of a function call is the result of applying the function to its arguments. If the arguments are not of compatible types, the resultant type will be an error.

It is perfectly acceptable for the result of a function call to be a function type.

5.3 Unary Operators

Unary operators are followed by a single expression and have a specific precedence (but which will always be lower than functions and parentheses).

Otherwise, they are identical to a function call of one argument.

5.4 Binary Operators

Binary operators are placed between two expressions and have a specific precedence (but which will always be lower than functions and parentheses). Precedence and associativity will be used to determine the exact meaning of an expression.

Otherwise, they are identical to a function call of two arguments.

5.5 Parenthetical Expressions

It is possible to group an expression in parentheses `( . . . )` in order to make the expression so enclosed have the precedence of a single term. The value and type of the expression remain unchanged.

6 Structs, Entities and Unions

In addition to primitive types, lists and hashes, FLAS offers the ability to build composite types.

Three basic methods of composition will be discussed here: `struct`, `entity` and `union`.

6.1 `struct`

A `struct` allows a set of values to be combined into a single value in a well-defined and type-safe manner.

```
                                    (10)  struct-declaration
                            ::=     STRUCT type-name
                                    poly-var*
(14)  struct-field-decl ::=     type-reference
                                    var-name
                                    struct-initializer?
```

Each of the field definitions in a `struct` defines a slot for a single value of a named type. The field name may be used to extract the value from the `struct` value later.

All values constructed using this method have the type of the `struct`.

`struct` definitions may be polymorphic. In this case, one or more polymorphic type variables may be specified after the `struct` name and may then be used in the definition of the fields.

`struct` fields may be of any type, including recursive references to the type being defined or to other type definitions which reference the `struct`.

Individual fields in the `struct` may have initializers. An initializer is used to populate the value of the field when the `struct` is created, although it may be overridden by creating the struct with a hash value.

6.2 `entity`

An `entity` is similar to a `struct`, but it has a unique *identity*.

```
(11)  entity-declaration
::=    ENTITY type-name
      poly-var*
```

Because they have a unique identity, `entity` values may be stored reliably in data stores, updated, referenced and retrieved.

The actual identity of the entity is an implementation detail not exposed to programs.

6.3 `union`

A `union` represents the set of values contained in the union of a collection of other types.

```
(16)  union-declaration ::=    UNION type-name
```

A `union` may be polymorphic. In this case, one or more polymorphic type variables may be specified after the `union` name and may then be used in the definition of the fields.

7 Crobags

`Crobag`s are a collision-resistant ordered bag of pairs of keys and entities.

They can be persistently stored with an identity. If so, managing the identity is an implementation detail.

A `Crobag` is a polymorphic type and is constrained by the type of entity which it contains

The keys are always strings. They may be encoded in UTF-8 except for the first character which must always be ASCII #21-#7E.

- It may be helpful to think of a `Crobag` as a map, but they are richer than that. It resembles a map in that internally, its keys are unique, each of which has a value, but it is possible to add duplicate keys in such a way that they are made unique during addition.
- The restriction on the opening character of the key is to ensure that `!` (#21) and ~ (#7E) can be reserved for the favorites feature. Note that the key is only used internally and is never displayed to users.
- The contents of a `Crobag` must always be entities because the `Crobag` is represented on a server as a list of entity `id`s.

7.1 API

A `Crobag` has semantics as if it were an `Object` definition.

Entities may be added to, removed from or replaced in the `Crobag` by methods.

`ctor new`

A `Crobag` can be created using the constructor `new`.

`method put key value`

It is possible to place an entity in a `Crobag` with a specific key by using `put`. This guarantees that the value associated with the specific key is the specified entity until it is changed

- Note (as in the commentary following) that a key property of `Crobags` is *collision resistance*. That is, if two clients make similar, but different, changes, the `Crobag` will resolve the inconsistency. The `put` method is intentionally *not* collision resistant: the change that arrives at the store of record last will determine the ultimate value of the entry associated with the key.

- □

`method insert key value`

It is possible to place an entity in a `Crobag` somewhere around the specified key using `insert`. This guarantees that a new entry will be created in the `Crobag` with the specified value and it will sort with other entries with similar keys.

-

Being precise about exactly what happens in a highly distributed, parallel system is always difficult. In this case, the system will eventually come to a consistent state where there is a new entry in the `Crobag` with the specified value. It will either have exactly the requested key or it will have a key whose prefix is the specified key.

Because there are no particular limits on what keys clients will choose to use, there is no guarantee that there will not be other entries with different intended keys interspersed with the entries for this intended key.

Consider:

bag.insert "pre" e1

bag.insert "pre" e2

bag.insert "prefix" e3

The system guarantees that the entries for `e1` and `e2` will start with the string `"pre"` but makes no other guarantees. It could choose to use the exact string `"pre"` for `e1` and `"pre_1"` for `e2`. Likewise, it could use `"prefix"` exactly for `e3`. In this case, the entries would sort as follows:

pre -> e1

prefix -> e3

pre_1 -> e2

which, while it might not be what you would expect, respects the concept that `e1` and `e2` "sort together": that is, all three appear in the set of entries beginning with `"pre"`.

- □

```
method upsert key value
```

It is possible to ensure that the entity in a `Crobag` associated with the specified key has the given value using `upsert`. If the key already exists, the fields in `value` will be used to update the current entity; if the key does not currently exist, the entity itself will be inserted precisely at the key.

- The semantics of `upsert` again need careful attention to detail. In an event driven system, the operation described may be applied multiple times (once on the client and once on each replicated server). The same semantics need to be applied on each occasion and the system needs to be eventually consistent.

  Consider a case in which two clients attempt to `upsert` the (previously absent) key `"welcome"` at approximately the same moment (sufficiently that neither sees the other's update before communicating their own message to the server).

  In each client, there is no such entry in the `Crobag` when the operation is performed locally. Consequently, the specified value is inserted at the specified key. Both clients then report to the server that the new (key, value) pair should be `upsert` ed.

  One will be processed first and will again be treated as a `put` operation and the entity - with its internally managed `id` - will be placed in the `Crobag`. When the second operation arrives, it will attempt to update the entity. Note that this update will happen regardless of whether the `id` or `version` matches.

  If the intent is simply to update the value of an entity already known to be in the `Crobag`, updating and saving the value is sufficient - the `Crobag` already has the entity's `id` and that is all it actually stores.

Commentary

7.1c A Language Feature

In spite of the brevity of this section, `Crobag`s are a language feature - and not just part of the standard library - because the runtime (on both client and server) is intimately aware of them and handles them as special cases. If they did not exist, it would not be possible to replicate their functionality in application code.

7.2c Use in Templates

Because `Crobag`s are ordered, they can be used as lists. Inside `card` definitions, it is possible to assign a `Crobag` value to a `container`.

When used in a template, the `Crobag` will respond to user gestures to select the appropriate entries to display based on a sliding window and provide infinite scroll to the container.

For `Crobag` which support arbitrary ordering, the container will also support dynamic reordering using drag and drop.

7.3c Collision Resistance

Standalone FLAS programs fail to realize much of the benefit of using a `Crobag`. The key benefits come from their properties of collision resistance and client-side caching.

Collision resistance is a property of collections which says that when two clients attempt to perform operations without having seen the results of the previous clients' operations, the server is able to process both operations in either order and come out with consistent results.

For comparison, the act of storing "the current contents of this list" on a server is not collision resistant because a change by one client would simply overwrite the changes of the other. Likewise, mindlessly replicating the operations on a list (such as "delete the fifth element") might not perform the right operation if the elements have changed order (for example because of an insert) before the operation is processed on the server.

`Crobag`s are collision resistant because if two clients attempt to perform operations at the same time, the server will resolve the apparent contradiction by taking into account the clients' intentions as specified by the operations they chose to use and then notify both clients as to the outcome. Because FLAS and Ziniki are notification based, both clients will end up with a consistent view of the entire `Crobag`.

As noted in the API section, it is important to understand the semantics of the individual operations to ensure that the correct intention is adequately described. The API has been carefully designed to give managed and expected behaviors, but it is important to choose the right operation for the situation in mind.

7.4c Client-Side Caching

`Crobag`s are often used in scenarios where the total contents of the `Crobag` can be vast, approaching infinite. Examples might be the contents of email folders, archives of newspaper articles, or historical stock prices.

In all these cases, it is possible to store *all* the data on a (sufficiently large) server cluster; it is not generally feasible to store it on - or transfer it to - a client.

`Crobag`s are capable of storing a `window` of data elements which the user is interested in and leaving the rest on the server to be dynamically loaded in response to a query or scroll event.

Clients cannot tell if a `Crobag` is stored persistently or transiently. Examples of transient `Crobag`s are the results of queries (such as messages in an email folder from a particular individual). Such transient `Crobag` objects will always be backed by a persistent `Crobag`.

7.5c Natural and Arbitrary Ordering

`Crobag`s will frequently have a *natural* ordering - for example, stock prices might be ordered by date and ticker symbol. This enables the client to determine a unique key and use the `put` and `upsert` operations.

On other occasions, the contents of a `Crobag` might have no such ordering. In this case it is possible to ask the `Crobag` itself for keys at the beginning or end of the `Crobag` or somewhere between two existing entries. In these cases the `insert` operation should always be used.

7.6c Favorites

`Crobag`s have native support for favorites. In the case where an arbitrary ordering is used this will generally not be necessary, although it is still a possibility. However, when a natural ordering is used, any keys starting with `!` will automatically appear at the front of the list and any keys starting with ~ will appear at the end of the list.

There is nothing magic about this: these characters are simply at the beginning and end of the range of acceptable key starting characters and are guaranteed not to be generated by the `start` and `end` key generating methods, but are generated by the `firstFavorite` and `lastFavorite` methods.

7.7c Events

All the `Crobag` operations are methods but in accordance with FLAS method semantics, do not directly update the state of the `Crobag`. Instead, they generate events which cause the `Crobag` to be updated immediately after the current method ends; the same events are then sent over to the server for processing there.

Additional internal events are sent from the server to the client to inform it of changes performed for consistency purposes (for example, if a key allocated for use with `insert` had already been allocated by another client). These are handled internally without reference to the application code, which does not need to be aware of them.

7.8c Usage Patterns

8 Functions

Functions map a set of values to a single result value.

```
                    (27)  function-case-definition
                          ::=
                                simple-function-case-definitio
                          |
                                degenerate-guarded-function-ca
                          |
                                guarded-function-case-definiti
               (28)  simple-function-case-definition
                          ::=    var-name
                                argument-pattern* EQ
                                expression
(29)  degenerate-guarded-function-case-definition
                          ::=    var-name
                                argument-pattern*
```

Functions are defined as a set of cases, each of which identifies a pattern for each of the input values it accepts and a result value if those patterns match.

If no patterns are specified, then the function is a *constant*, and only one case may be specified.

If more than one case is defined, each case must specify the same (non-zero) number of patterns.

The cases may be presented in any order, and the patterns may overlap arbitrarily, but for each possible combination of values exactly one case must be a unique *best match*. If guards are used, the value of the function will be determined by evaluating the guards for this best match case as below; otherwise, the single equation associated with this best match case will be used to determine the value of the function.

The type of a given function is determined by inference as discussed in the next chapter.

8.1 Pattern Matching

In each case of the function, each formal function argument is specified as a pattern. FLAS supports three types of patterns.

```
(36)  argument-pattern  ::=
                              argument-pattern-variable
                        |      argument-pattern-list
                        |      argument-pattern-typed
                        |      argument-pattern-ctor
                      (38)  argument-pattern-variable
                        ::=    var-name
                           (39)  argument-pattern-list
                        ::=    OSB CSB
                        |      OSB argument-pattern
                               comma-argument-pattern*
                               CSB
                          (40)  comma-argument-pattern
                        ::=    COMMA argument-pattern
                          (41)  argument-pattern-typed
                        ::=    ORB type-reference
                               var-name CRB
                           (42)  argument-pattern-ctor
                        ::=    ORB type-name OCB CCB
                               CRB
                        |      ORB type-name OCB
                               field-argument-pattern
                               comma-field-argument-pattern
                               * CCB CRB
                          (43)  field-argument-pattern
                        ::=    var-name COLON
                               argument-pattern
                    (44)  comma-field-argument-pattern
                        ::=    COMMA
                               field-argument-pattern
```

A simple *variable* pattern consists of just a variable name which has not been previously defined in this scope. This specifies no new information about the argument.

A *typed* pattern must be written in parentheses with a single type reference and a single variable name. The variable is restricted to being of the designated type.

A *constructor match* pattern matches some or all of the fields of a `struct` or `entity` definition. If the `struct` or `entity` has no arguments, then the constructor name serves as a match by itself. Otherwise, the pattern must be enclosed in parentheses and the constructor name is followed by a hash construct which matches field names to sub-patterns. By itself, a constructor match pattern constrains the types that the case can handle but does not introduce any new variables into the scope. However, any nested patterns are constrained to belong to the appropriate types for the fields they match.

8.2 Guarded Equations

In addition to pattern matching, it is possible to choose between expressions by using guards.

```
                    (30)  guarded-function-case-definition
                            ::=    var-name
                                   argument-pattern*
(31)  guarded-equations     ::=    guarded-expression+
                                   guarded-default-expression
                                   ?
                                   (32)  guarded-expression
                            ::=    GUARD expression EQ
                                   expression EOL
                          (33)  guarded-default-expression
                            ::=    EQ expression EOL
```

In order to use guarded equations, the function name and patterns must be presented on one line and the guarded equations in a nested block.

The last guarded equation may be just an equation introduced by (=), and not containing a guard. Otherwise, a guarded equation consists of a guard, introduced by (|), and an equation introduced by (=).

Each of the guards must be of type `Boolean`.

Each of the guards is evaluated in the order presented until one of them evaluates to `True`. The value of the function is then defined by the corresponding equation.

If none of the guards evaluate to `True`, the default equation will be used if present. If no default equation is present, the value of the function is an `Error`.

8.3 Function Nesting

Function cases may include a nested scope consisting of the immediately following lines indented one tab deeper.

Each case may have its own nested scope. Nothing is shared between these scopes. When guarded equations are used, the nested scope must appear after the final equation nested a further level of indentation. The nested scope is shared across all the equations of the case.

The nested scope may define functions, tuples, standalone methods and `handlers`. These definitions are only visible to the enclosing function and definitions in the nested scope.

Definitions in the nested scope may not define names that are defined in a containing scope.

All the definitions in all containing scopes are visible to definitions in the nested scopes, including parameters defined in patterns.

8.4 Tuple Definitions

Tuple definitions allow the elements of a tuple value to be extracted into a scope.

```
(34)  tuple-definition  ::=   ORB var-name
                              comma-var-name+ CRB EQ
                              expression
(35)  comma-var-name    ::=   COMMA var-name
```

A tuple definition is identical to a set of parallel constant function definitions. No patterns are permitted. The expression and inner scope of the tuple definition may not refer to the names defined by the tuple definition.

8.5 Standalone Methods

Standalone methods are pure functions defined using method syntax.

```
                          (57)  standalone-method-definition
                          ::=     METHOD
                                  method-definition
                          (58)  object-method-definition
                          ::=     METHOD
                                  method-definition
(59)  method-definition   ::=     var-name
                                  argument-pattern*
```

The method syntax is covered in another place.

8.6 Functions with State

Functions defined within actors (`cards` and `agents`) may access the state members of the actor.

Functions defined within `handlers` may access the lambda values of the `handler`.

Commentary

8.1c Evaluation

The specification above describes formally how a FLAS program should be understood from a mathematical perspective. However, it is also important to understand how evaluation is actually performed.

The expression results of functions do not perform any evaluation - they simply record that a function or operator will be applied to a set of sub-expressions at some later time. This expression construct - called a *closure* - is the physical return value of a function. In order to be evaluated, this must be passed to a mechanism that will determine its value.

FLAS is a *lazy* functional language because it has the ability to create such expressions that will never actually be evaluated.

Evaluation of an expression happens in one of two cases: either if the value is used "at the top level" (in an initializer, a template, a guard or as the return value of a method called from the container) or if the value is subjected to pattern matching.

The vast majority of evaluations in FLAS are driven by pattern matching.

Pattern Matching

In order to determine which case of a function to apply, it is necessary to resolve enough of the structure of the various arguments to make an unambiguous decision which of the cases applies.

The first rule is that for every value there must *be* an unambiguous correct decision. The compiler will generate an error if two cases are identical. For example, given:

```
f 0 = 1

f 0 = 2
```

It is impossible to unambiguously define the value of `f 0`.

However, it is acceptable to have cases overlap provided that one case is clearly "more precise" than the other, regardless of the order in which they are presented. For instance, with:

```
g 0 = 1

g (Number n) = 2
```

It is possible to say that `g 0` has the value `1` and `g 0` has the value `2`. This would be the same for the function `h` with the cases defined in the other order:

```
h (Number n) = 2

h 0 = 1
```


Only as much of the structure of the argument as is required to determine the case needs to be evaluated. For instance, it is possible to determine the emptiness or not of a list without determining the whole list:

```
isEmpty Nil = True

isEmpty (Cons h t) = False
```

Even if applied to an infinite list, say `isEmpty allPrimeNumbers`, this is able to complete in finite time because it is *not* necessary to evaluate all the prime numbers before determining if there are any (the first is sufficient).

It is *not* a compile-time error to have some cases by uncovered. For example, with:

```
k 0 = 1

k 1 = 2
```

It is possible to determine that the case `k 2` is not covered. This will, however, generate a runtime error if `k 2` is ever evaluated.

It is a compile-time error to have an argument with an unclear type. This is discussed in the next chapter.

Guards

Guards are processed in order. Each guard that is processed must be fully evaluated to either return `True` or `False`. However, when a guard returns `True`, the corresponding expression is selected and no further guards will be evaluated. Because of this, thought should be given not just to the correct logic in deciding the order of guards but also to the relative cost of evaluation (if this can be determined).

8.2c Scoping

The rules concerning scoping may seem complex at first, but they are well founded.

The idea is to permit complex expressions to be broken down by use of locally named subexpressions. It is very common in scoped definitions to use constant definitions which abstract some of the complexity of the main calculation. In general, however, these are not actually constant because they depend on names inherited from enclosing scopes.

The rules regarding names are simply to ensure that there is no ambiguity about what a name means. In any scope where a name appears, it must have exactly one meaning. Given that all names from the outer scope - and all names defined in the patterns of the current case - are incorporated into a nested scope, it is not permitted to define new functions or parameters reusing these names since that would create an ambiguity.

8.3c Standalone Methods

Although standalone methods *look* like contract or handler methods, they are fundamentally different. In reality, all methods are somewhat illusory - they are a function mapping arguments to messages. But for most methods, they must conform to a specified contract and their values are immediately interpreted by the system. These are fundamental language features.

On the other hand, standalone methods are just there as syntactic sugar to make it easier to define functions returning a list of messages. They may have an arbitrary number of cases and patterns and perform pattern matching in the same way as functions. They may be nested inside methods and functions.

9 Type Checking and Inference

FLAS is a strongly typed language. Every value is defined as belonging to a specific type and these types are tracked during compilation and at runtime.

FLAS tries to minimize the number of type declarations, requiring them in type declarations and at code boundaries (in `contracts` and `handlers`).

Some contexts require values of specific types. In these situations, the type of the value will be determined and an error emitted if it is incorrect.

In all other situations, the existing type information will be evaluated and, if possible, the types of remaining symbols will be determined. If no determination can be made, an error will result.

9.1 Type Declarations

In `struct` and similar declarations, each field is given a name and a type. Whenever the field is subsequently referenced, it will be of the specified type.

In `contract` method declarations, each formal method argument must be a *typed pattern*. All implementations of this contract method must have the same number of arguments, each of which has the same type as that defined by the corresponding contract argument.

9.2 Type Checking

Values used in equation guards must always be of type `Boolean`.

Values placed in template content cells, or used as template styles, must be of type `String`.

Values used in template conditional styles must of type `Boolean`.

Values returned from methods must be `Message`, `List[Message]` or a compatible value.

9.3 Type Inference

The types of function, standalone method, tuple and parameter values can be inferred from the context. This is done automatically and the resultant types stored and, if applicable, exported.

The most general type - including polymorphism - will be calculated.

If no single type can be deduced, an error will result indicating the set of symbols which seem to have mutually contradictory type expectations.

9.4 Overriding the Type Mechanism

The `cast` operator acts as a function of two arguments. The first argument is the desired type; the second argument is a value of unknown or wider type. The result is the same value but of the desired type.

The actual type will be preserved at runtime and will be checked against the desired type. A runtime error will be generated if the value is not of the specified type.

-

In general, the `cast` operator is used when a value has type `Any` due to circumstances outside the program's control. Some contracts - not knowing the inner workings of the program - specify that values are of type `Any` because they cannot do better. Some data structures - such as `Hash` and `List[Any]` - return values of type `Any`.

In all these cases, `cast` is the simplest and most reliable way of bringing the value back into the type safe world. It should be used as close to the offending interface as possible.

Using `cast` in most other circumstances should be considered a code smell, and is likely to cause runtime errors.

-

-

I'm not really sure how much more to actually say in the specification.

-

Commentary

9.1c Type Inference Algorithm

Type checking is easy. Type inference is hard.

Type inference in FLAS is based on the Hindley-Milner algorithm. The algorithm works from what is known to what is unknown. The first step is to deduce the dependencies between names in the program and break all the definitions into groups of functions which depend only on either what has gone before or on each other.

Each such group of functions will be tackled together, along with all of the formal parameters deduced from patterns.

Nested functions cause a particular problem, especially when (as they usually do) they reference parameters from an enclosing scope.

The essence of the algorithm is to determine the input types of a function by taking a union of all the patterns. Complicated by having union types. Try and find the minimal union type which fits. Can be helped/resolved by having the "catch-all" case specify the type you want rather than just having a variable.

Can use "Any" to say you want to accept everything.

Can accept "Error" and not have it show up in the type.

The output type of the function is handled by dealing with each expression in turn and figuring out what type it is, and then doing a minimal union of all those values. `Any` is not an acceptable answer unless one of the cases resolves to type `Any`.

This is fine if the group has just one function depending only on things that have been previously defined. It gets more complicated when the group has multiple members.

In this case, a *type variable* is introduced for each unknown in the group:

- each function name
- each standalone method name
- each tuple variable name
- each parameter name that is not clearly typed
- each polymorphic type used in the definitions of the function parameters.

Note that a single polymorphic type name used within a single function definition maps to the same type variable. But if the same name is used in *different* function definitions, one type variable will be introduced for each function.

Once the type algorithm has processed, the type variables will have collected information about all the constraints that are required of them and a unification process is run which deduces the maximal types which fulfill all the constraints for all the variables. The resulting types are then inserted in the appropriate places to complete the type definitions.

If one or more of the type variables does not have a consistent solution, an error is reported indicating the problem and the usages that led to it. These messages are inherently complex, but it is hoped that over time they will become more easily comprehended.

10 Contracts

Contracts define the protocols by which actors communicate with each other.

Contracts come in three varieties: *implementation* contracts handle "downward" or "nested" communication to contained *agents* or *cards*; *service* contracts handle "upward" or "outward" communication from contained actors to service providers; and *handler* contracts define the structure of responses.

Contracts are defined with a kind, a name and a set of methods.

```
                                  (23)  contract-declaration
                            ::=     contract-intro
                                  type-name
(24)  contract-intro        ::=     CONTRACT
                            |       CONTRACT SERVICE
                            |       CONTRACT HANDLER
```

10.1 Contract Methods

Every contract method has a name.

Contract methods may have arguments. Each of these needs to be a *typed pattern* indicating the expected type and function of the parameter.

```
                                  (25)  contract-method-decl
                            ::=     OPTIONAL? var-name
                                  argument-pattern-typed*
                                   handled-by?
(26)  handled-by            ::=     HANDLE
                                  argument-pattern-typed
                              (41)  argument-pattern-typed
                            ::=     ORB type-reference
                                  var-name CRB
```

A contract method may optionally accept a *handler* of a specific type over which responses will be sent. The handler's type must be a contract.

Commentary

10.1c Role of Contracts

Contracts are absolutely central to the operation of FLAS.

Cards, agents and services are essentially unaware of each other and are completely decoupled except through contracts and entities.

While it is possible for one card to create another card directly by name, much of the time cards are created because an entity is placed in a punnet. The only way in which a containing card knows how to interact with the contained card is through a set of contracts it expects the cards in that punnet to implement.

10.2c Directionality

It is fairly easy to see the directionality in cards: there is one card that "is" the application, and that contains other cards, which contain other cards… The application card is itself in a client-side container. One set of messages ("do this" messages, if you will) flow downwards, while another set of messages ("request" messages) flow the other way, looking for services.

If no cards or agents provide a service being requested, the request is propagated to the client-side container. Some contracts have services implemented directly in the container (such as `Repeater` and `Ajax`). If the client-side container cannot find an implementation, it may forward the request to a server-side container if one is connected.

10.3c Testing

[1] This feature is not yet implemented.

11 Objects

Objects in FLAS provide the ability to encapsulate data and methods in a single structure.

Objects can only be created within the scope of an actor.

Objects may be used to encapsulate entities, thus giving the (data-only) entity the appearance of being a persistent object.

```
                                    (71)  object-declaration
                              ::=    OBJECT type-name
(72)  object-scope-unit       ::=    state-declaration
                              |      object-ctor-definition
                              |
                                    named-template-definition
                              |      object-acor-definition
                              |
                                    object-method-definition
                              |
                                    function-case-definition
                              |      handler-definition
```

11.1 Object State

Objects may have a state definition.

The fields in the state definition may be initialized provided that the initialization code does not have any external dependencies or generate any messages. All other initialization code must be put in constructors.

The state members are only visible to code definitions within the object. Tests have special permission to access state fields using the `assert` and `shove` operations.

```
(73)  state-declaration ::=    STATE
```

11.2 Object Constructors

Objects are constructed using named *constructors*.

An object must have at least one constructor.

```
(74)  object-ctor-definition
::=   CTOR method-definition
```

The constructor definition is preceded by the keyword `ctor`.

A constructor has a name and zero or more arguments

The creation of a new object is implicit; the constructor method returns messages which initialize it.

If no initialization is necessary, the constructor may have an empty body.

The constructor is called by using the type name, followed by a `DOT` (`.`), followed by the constructor name and any arguments.

Because constructors may return messages along with the new object, they may only be called from contexts in which message methods are allowed.

11.3 Object Methods

Objects may declare read-only methods, called *accessors*. Accessors are pure functions that map values to values. They may reference state members.

Objects may declare update methods which return messages. Update methods may wish to also return a value. In this case, the method must return a value of type `ReturnWithMessages`.

```
(75)  object-acor-definition
::=   ACOR
      function-case-definition
(58)  object-method-definition
::=   METHOD
      method-definition
```


These definitions are accessible to clients of the object using the member expression syntax: a variable name (of the appropriate object type), followed by `DOT` (`.`), followed by the accessor or method name.

Accessors may be referenced from any context; methods may only be referenced from a message method context.

11.4 Services

11.5 Nested Scope

Objects may nest arbitrary functions and handlers.

These nested definitions are only visible to definitions within the object definition.

These definitions may access the object state.

Commentary

11.1c Not an Object-Oriented Language

Although it contains the `object` definition, FLAS is not, and makes no claims to be, an object-oriented language and you will suffer heartache if you try to treat it as one.

The main building blocks in FLAS are actors, and their primary means of communication is through contracts. The main data representation structures are (transient) `structs` and (persistent) `entitys`. Objects are an adjunct to this to make some abstractions and encapsulations easier to write and maintain.

In general, `objects` are written as wrappers around `entitys`, thus providing the appearance of behavior on something which is really just data.

12 Methods

Methods are syntactic sugar for functions that map values to messages.

Methods may exist in a number of scopes and this determines how they are introduced.

In each case, the body of the method is the same.

```
                                (60)  method-guards-or-actions
                                ::=     method-guards
                                |       method-actions
(61)  method-guards             ::=     method-guard*
(62)  method-guard              ::=     GUARD expression
(63)  method-actions            ::=     method-action*
(64)  method-action             ::=     message-method-action
                                |       assign-method-action
                                   (65)  message-method-action
                                ::=     SEND service-method
                                        expression*
                                        maybe-handled? EOL
                                |       SEND expression EOL
(66)  service-method            ::=     var-name APPLY
                                        var-name
(67)  maybe-handled             ::=     HANDLE var-name
                                    (68)  assign-method-action
                                ::=     member-path SEND
                                        expression EOL
```

12.1 Guards

Methods may be guarded. Guards in methods work exactly as in functions.

12.2 Sending Messages

12.3 Updating State

Commentary

12.1c Messages

FLAS is a pure functional language wrapped in an actor model. Each interaction represents a mapping from one system state to another, and this can be expressed as a set of messages. During the evaluation of the interaction, *the actor state does not change*, no actor has access to the internal state of any other system component and the state of an actor never changes *except through an explicit interaction*. In this way, the apparent state of an actor is always consistent and it is possible to reason logically about both the actor state and the overall system state.

FLAS can be considered to be *transactional* in that the interaction either generates a set of messages or it does not, and then either all of those messages are consumed or none of them are. Of course, it takes more than that to make a system truly transactional in an ACID sense

Messages are just values. The `Message` type is a `union` consisting of various nested message types. A function can create values of these types.

A method is simply a function that is called from a context in which it is possible to directly harness the messages being returned. There are three such locations:

- within implementations of contracts inside actors;
- within the event handlers of cards or objects;
- within any existing method declaration.

Standalone methods are exactly syntactic sugar for moving between methods and functions and may only be called from other methods or functions.

12.2c Conflicts

It should not be possible to attempt to update the state of an object or actor inconsistently. It should, however, be apparent that it is possible:

object Obj

state

Number n <-0

method wrong

n <- 1

n <- 2

And, indeed, in this trivial case, it is easy to spot (even at compile time), but it is clear that there are more complex cases where more subtle conflicts can arise, including updating a value nested within a value which is removed from the actor state by another update.

It is anticipated that as time passes, the algorithm for detecting such conflicts (especially at compile time) will be improved, but in the meantime it is important to try and write code to minimize conflicts.

13 HTML Visual Designs

In order to generate HTML from the template mechanism (see §12), it is necessary to indicate within the HTML design where the `card` and `item` boundaries are, and how the fields are mapped within these.