International Frogans Address Pattern - 1.0

Abstract

This document sets forth the pattern applicable to Frogans addresses. A Frogans address is a string of characters used to identify a Frogans site published on a computer network, such as the Internet or an intranet. A Frogans address may contain international characters and may be written either from left to right or from right to left, depending on the writing system.

Status

This document is an official technical specification of the Frogans technology.

This technical specification was adopted by the OP3FT on March 5, 2014.

Comments on this document are welcome and may be made on the Frogans technology mailing lists, accessible at the following permanent URL: https://lists.frogans.org/.

Location

This document is accessible at the following permanent URL: https://www.frogans.org/en/resources/ifap/access.html.

Copyright Statement

This document must be used in compliance with the Frogans Technology User Policy, accessible at the following permanent URL: https://www.frogans.org/en/resources/ftup/access.html.

Table of Contents

1. Introduction 3
1.1. Background 3
1.2. Purpose 4
1.3. Intended audience 5
1.4. Stability and security 6
1.5. Compliance 6
2. Terminology 8
3. Frogans address strings 10
3.1. String character set 10
3.2. String formation 12
3.3. Eligible characters 14
3.4. Directionality 15
4. Structure of a Frogans address 18
4.1. Asterisk character 18
4.2. Network name 18
4.3. Site name 19
4.4. Connector characters 19
5. Generating the reference form of a Frogans address 21
6. Evaluating the length of a Frogans address 23
7. Checking whether two Frogans addresses are identical 24
8. Usage of ASCII-encoded Frogans addresses 25
9. References 28
9.1. Normative references 28
9.2. Informative references 28
Appendix A. IFAP lookup tables 30
Appendix B. Pseudocode syntax 34
Appendix C. Assistance in implementing the specification 38
C.1. String character set 38
C.2. String formation 39
C.3. Eligible characters 54
C.4. Directionality 55
C.5. Structure 61
C.6. Reference form 67

1. Introduction

1.1. Background

Started in 1999, the Frogans project aims to introduce a new software layer on the Internet alongside other existing layers such as E-mail or the Web. The goal of this new software layer, called the Frogans layer, is to enable the publishing of Frogans sites.

The Frogans technology developed for the Frogans project is the foundation of the Frogans layer. It includes an addressing system allowing users to access each Frogans site via a unique Internet address, called a Frogans address.

A Frogans address is an identifier. It is made up of a string of characters.

At the time the original Frogans address pattern was designed, the goals were to define a pattern with the following characteristics:

* It had to be short and simple.

* It could not contain any technical information.

* It had to clearly stand out in various contexts where Frogans addresses could appear (such as in a printed document, on a business card, or when displayed as a link on a Web page or in an E-mail message).

* It had to be original so that users could easily distinguish Frogans addresses from other Internet addresses (such as those pointing to Web sites or to content published on other software layers which may be introduced on the Internet in the future).

URIs [RFC3986] and domain names [RFC1034] were not chosen as a basis for Frogans addresses as they could not directly achieve these goals without modifying their scheme or syntax.

The original pattern chosen to achieve these goals is described in the Frogans Network System Language specification released in 2004 [FNSL].

The original Frogans address pattern defines a name space with the following features:

* The name space uses two main levels.

* The two levels are separated by a distinctive sign: the asterisk character.

* The first level designates the Frogans network, i.e. the group that the Frogans site belongs to.

* The second level reflects the content of that Frogans site.

The original Frogans address pattern was intended to support the ASCII character set [ASCII] only. Frogans addresses were read from left to right. The first level in the Frogans Address always appeared on the left.

1.2. Purpose

The purpose of this document is to set forth a new pattern applicable to Frogans addresses.

Since the creation of the original Frogans address pattern, the use of the Internet has continued to expand worldwide. Thanks to the widespread adoption of technologies such as the Unicode Standard, the use of international characters has been generalized. They are now used extensively both for the content exchanged over the Internet (such as E-mail messages and Web pages) and for domain names through the development of Internationalized Domain Names (IDNs).

In order to meet the needs of users worldwide, the original Frogans address pattern must be extended to support international characters so it is no longer limited to the ASCII character set [ASCII]. This includes the support of both left-to-right and right-to-left writing systems.

Extensive work has already been carried out on international identifiers, including Internationalized Domain Names (IDNs), by organizations such as the Unicode Consortium, the World Wide Web Consortium (W3C), the IETF, ICANN, and various domain name registry operators. The work reflects the many lessons learned about security issues in systems supporting multiple languages, and how to mitigate them. The new Frogans address pattern obviously needs to build upon these achievements.

The new Frogans address pattern must retain the characteristics of the original Frogans address pattern as well as the features of its name space.

The new Frogans address pattern must also remain backward compatible with the original pattern, with two exceptions. First, the lengths of the two main levels (referred to as the network name and the site

name in this document) have been harmonized to share the same minimum and maximum values. Second, in order to avoid confusion with domain names on the Internet, the full stop character (.) has been eliminated from the second level.

It is important to note that the addressing system used for Frogans sites is not intended to replace domain names nor the Domain Name System (DNS). In fact, it operates on top of the DNS via a specific generic top-level domain (the .frogans gTLD), and on top of other core Internet protocols and standards. The functioning of this addressing system is described in the Frogans Network System Language specification [FNSL].

1.3. Intended audience

This document is intended for those involved in the Frogans address registration process, such as Frogans address holders, FCR account administrators, and the Operator of the Frogans Core Registry (FCR).

This document is also intended for developers wishing to implement software using Frogans addresses, and in general for anyone interested in the addressing system used for Frogans sites.

To comprehend the choices made in this specification, it is necessary to understand the context in which these choices are made. This is not an easy task, since the multiple standards and specifications underlying the Frogans address pattern require time and effort to assimilate and use correctly.

Therefore, in order to make this specification accessible to the widest possible audience, it was decided to provide, when required, relevant background information before describing the choices made. As a result, this specification often alternates background information and rules applicable to Frogans addresses. The background information may include a detailed reference to the underlying standard or specification.

In addition, the appendices provide assistance in implementing certain parts of this specification. They contain lookup tables with pre-processed lists of code points (Appendix A), pseudocode syntax (Appendix B), and a series of verification and generation processes (Appendix C). The goal is to avoid the need for developers to access and analyze the data and the algorithms defined in the multiple standards and specifications involved in the Frogans address pattern.

1.4. Stability and security

An important difficulty must be overcome when specifying international identifiers.

When a security issue is discovered in one or more specifications concerning international identifiers, the specifications in question should be amended to mitigate the problem. However, widely distributed and installed implementations should remain compatible, as they would be difficult to update in a reasonable delay.

To solve these contradictory requirements, the OP3FT Bylaws [BYLAWS] call for the creation of a separate technical specification dealing with security issues, notably concerning support for multiple languages. This specification is called Frogans Address Composition Rules (FACR).

Thus two specifications apply to Frogans addresses: IFAP (this document) and FACR. They play complementary roles:

* IFAP defines Frogans addresses from a technical standpoint. FACR focuses on security rules.

* IFAP is designed to be language-independent. FACR covers language-related issues.

* IFAP provides a stable base that is intended for the long term. FACR will be updated as needed to deal with new security issues.

* IFAP is to be implemented globally in all software using Frogans addresses. FACR is to be implemented solely by the FCR Operator.

The rules in FACR are enforced by the FCR Operator at the time a Frogans address is added to the FCR. The rules in FACR are applied to Frogans addresses that are already compliant with the IFAP specification.

This two-part model for specifying Frogans addresses combines the stability required for a widely distributed and installed technology with the flexibility and reactiveness demanded to solve security issues that may arise.

1.5. Compliance

The rules applicable to Frogans addresses in this specification are defined in succession. The definition of each rule assumes compliance with all preceding rules.

A conforming implementation of this specification is an implementation which is compliant with all descriptions appearing in this document, except for:

* descriptions in paragraphs that do not directly concern the Frogans technology, but provide background information intended to help understand the context and the reasons for choices made

* descriptions found in sections that are indicated as not normative, such as the appendices which provide assistance in implementing certain parts of this specification

* descriptions in the form of examples that illustrate certain aspects of the specification

Hence, unlike in specifications elaborated by several other organizations, requirement levels in this specification are not indicated using key words such as "must", "must not", "should", and "should not" defined in RFC 2119 [RFC2119]. This applies to all specifications elaborated by the OP3FT.

Normative and informative references appear between square brackets [] in this document. Their details are included in the References section.

2. Terminology

This section defines key terms used in this specification.

OP3FT

> A non-profit organization whose purpose is to hold, promote, protect, and ensure the progress of the Frogans technology in the form of an open standard for the Internet, available to all, free of charge.

Frogans technology

> A secure technology used to implement a new software layer on the Internet, alongside other existing software layers such as E-mail or the Web. The Frogans technology makes it possible to publish Frogans sites.

Frogans site

> A set of Frogans pages, called "slides", hyperlinked to each other, available online on the Internet or in an intranet, at a Frogans address. A Frogans site can be published by any individual or organization, from anywhere in the world, in any language.

Frogans address

> A string of characters serving as the identifier of a Frogans site. Frogans addresses include two parts, separated by the asterisk character: the network name and the site name. Frogans addresses may contain international characters and may include uppercase, lowercase, and accented characters. Frogans addresses may be written from left to right or from right to left. For example, in the left-to-right writing direction, the pattern of a Frogans address is "network-name*site-name".

Separator character

> The asterisk character. It is used to separate the network name and the site name in a Frogans address.

Network name

> The string of characters in a Frogans address that precedes the separator character when writing the Frogans address.

Site name

The string of characters in a Frogans address that follows the separator character when writing the Frogans address.

Connector character

A character that can be used to connect different words included in a network name or a site name.

Reference form

Form of a network name, a site name, or a Frogans address generated to evaluate its length and to check whether two network names, site names, or Frogans addresses are identical. This form is not intended for display to end users.

Preferred form

Form of a network name, a site name, or a Frogans address as registered in the Frogans Core Registry by its holder. Frogans Player uses this form to display Frogans addresses to end users.

Frogans network

A group of Frogans addresses that have the same network name.

Frogans Core Registry, FCR

The database which contains all registered Frogans addresses and Frogans networks. The database belongs to the OP3FT.

FCR Operator

The organization responsible for the technical and commercial operation of the FCR, under a delegation agreement with the OP3FT.

Frogans Player

Free-of-charge software used to browse Frogans sites. Frogans Player is to be made available on a wide range of fixed and mobile devices.

3. Frogans address strings

3.1. String character set

A Frogans address is made up of a string of characters.

In technical terms, a character string can be seen as a series of numbers, where each number corresponds to a specific character. This correspondence between numbers and characters is defined in a table called a "character set".

Historically, since the original ASCII character set [ASCII] which was designed for the English language, numerous other character sets have been defined over the years in various parts of the world in order to support other languages. For example: GBK for simplified Chinese, Shift-JIS for Japanese, the ISO-8859-xx series for other languages, etc.

To simplify the interoperability of computer systems worldwide, a character set was defined to include all the characters of all the world's languages. This universal character set is called Unicode [Unicode]. In the Unicode character set, the numbers corresponding to characters are called "code points". Code points are grouped into collections called "Unicode scripts", each one representing a writing system. A Unicode script can be used in the context of one or more languages.

The standard way of representing a Unicode code point is "U+code" where "code" is a series of four to six uppercase hexadecimal digits representing the numerical value of the code point. For example, U+96CD represents the code point of the character corresponding to "harmony, union; harmonious" in the Han Unicode script, which is used in the context of the Chinese, Japanese, and Korean languages.

A given language may make use of more than one Unicode script. For instance, the Japanese language makes use of three Unicode scripts: Han, Hiragana and Katakana.

Unicode provides the means to support both left-to-right and right-to-left text, as well as bidirectional text. Right-to-left text is used in the Arabic and Hebrew writing systems.

The code points in a Unicode string are in the order in which the text is written.

Storage or transmission of a Unicode string is achieved by encoding its code points into an array of bytes, using an encoding method such as UTF-8 [UTF-8] or UTF-16 [UTF-16].

In light of these extensive features, the Unicode character set has been progressively adopted in the information technology industry and is now widely used.

The character set used to represent Frogans address strings is the character set defined in version 6.3.0 of the Unicode Standard [Unicode], which is the latest available version at the time this specification is being completed.

This specification is tied to this version of the Unicode Standard, and in that sense it is not a "living standard". A new version of IFAP will be prepared if future corrections or enhancements to the Unicode Standard have an impact on the use of Frogans addresses. This would be the case, for example, if important code points were to be added or removed, or if their properties were to be modified. In any case, Frogans addresses will remain compatible with Frogans addresses defined under future versions of IFAP.

In this document, Frogans addresses are described using code points, irrespective of the encoding method used to store or transmit them. Each code point is represented using the "U+code" format described above, followed by its name in the Unicode character set. For example, the code point "U+0046" is represented as "U+0046 LATIN CAPITAL LETTER F".

The Unicode Standard defines fundamental classes of code points, referred to as General Categories (see the Unicode Standard, section 4.5 General Category) and as Basic Types (see the Unicode Standard, section 2.4 Code Points and Characters). The Basic Types are Graphic, Format, Control, Private-Use, Surrogate, Noncharacter, and Reserved.

Code points with the Basic Type of Control, Private-Use, Surrogate, Noncharacter, and Reserved are not suitable for use in identifiers since either their usage is meant to be defined outside the Unicode Standard or they are reserved.

Code points with the Graphic Basic Type correspond to letters, marks, numbers, punctuation, symbols, and spaces, while code points with the Format Basic Type are invisible but affect neighboring characters, or are line/paragraph separators.

Code points with the Basic Type of Control, Private-Use, Surrogate, Noncharacter, and Reserved cannot be included in Frogans address strings.

Code points with the Format Basic Type cannot be included in Frogans address strings, except for the following code points: U+200C ZERO

WIDTH NON-JOINER and U+200D ZERO WIDTH JOINER.

Following the rules presented in Section 3.1, a total of 109,977 code points and 102 Unicode scripts are available for use in Frogans address strings. Subsequent rules will reduce these totals.

For assistance in implementing a function to verify compliance regarding the string character set, see Appendix C.1.

Several additional rules applicable to the use of code points in Frogans addresses will be defined in subsequent sections of this specification. Such rules are needed because the Unicode character set was initially designed to manage general text rather than identifiers. These rules include, for example, string formation, eligible characters, and directionality.

Many of these additional rules are based on work by the Unicode Consortium and the IETF concerning the use of identifiers and the introduction of Internationalized Domain Names (IDNs) in the Domain Name System on the Internet.

3.2. String formation

The Unicode Standard [Unicode] [UAX15] defines four normalization forms for Unicode strings of characters (see the Unicode Standard, section 3.11 Normalization Forms). These normalization forms are Normalization Form D (NFD), Normalization Form C (NFC), Normalization Form KD (NFKD), and Normalization Form KC (NFKC).

Normalization form NFKC erases both canonical and compatibility differences, and generally produces a composed result. It is recognized as the most appropriate form for identifiers in the Unicode Standard Annex #31 [UAX31].

The normalization form used for Frogans address strings is NFKC. In other words, Frogans address strings are not modified when they are normalized to NFKC.

As a result, a code point which is modified through an operation that returns the code point normalized to NFKC cannot be included in a Frogans address string.

The Unicode Standard defines combining characters, which are used in sequence to combine with a preceding base character (see the Unicode Standard, section 3.6 Combination). Combining characters include characters such as accents, diacritics, Hebrew points, Arabic vowel signs, and Indic matras. For example, the U+0302 COMBINING CIRCUMFLEX ACCENT character is a combining character.

The General Category of combining characters is M (Combining_Mark).

Frogans address strings cannot contain more than 30 successive code points corresponding to combining characters.

The Unicode Standard defines combining classes that are used to determine which sequences of combining characters are to be considered canonically equivalent and which are not (see the Unicode Standard, section 3.11 Normalization Forms). Each code point is assigned a combining class, referred to as its Canonical_Combining_Class property.

The Unicode Standard also defines a joining type that is used to describe the cursive joining behavior of each character as it interacts with the cursive joining behavior of adjacent characters (see the Unicode Standard, section 8.2 Arabic). Each code point is assigned a joining type, referred to as its Joining_Type property. The Joining_Type property values are R (Right_Joining), L (Left_Joining), D (Dual_Joining), C (Join_Causing), T (Transparent), and U (Non_Joining).

The U+200C ZERO WIDTH NON-JOINER code point can be included in Frogans address strings only if one of the following two conditions is met:

* The U+200C ZERO WIDTH NON-JOINER code point is preceded in the Frogans address string by a code point with the Canonical_Combining_Class property value equal to 9 (Virama).

* The U+200C ZERO WIDTH NON-JOINER code point is included in a sequence of code points that matches the following pattern: a code point with the Joining_Type property value equal to L or D, followed by zero or more code points with the Joining_Type property value equal to T, followed by the U+200C ZERO WIDTH NON-JOINER code point, followed by zero or more code points with the Joining_Type property value equal to T, followed by a code point with the Joining_Type property value equal R or D. This sequence can be located anywhere within the Frogans address string.

The U+200D ZERO WIDTH JOINER code point can be included in Frogans address strings only if it is preceded in the Frogans address string by a code point with the Canonical_Combining_Class property value equal to 9 (Virama).

Following the rules presented in Section 3.2, applied in addition to the preceding rules, a total of 105,190 code points and 102 Unicode scripts are available for use in Frogans address strings. Thus the rules presented in this section eliminate 4,787 code points and zero

Unicode scripts. Subsequent rules will further reduce these totals.

For assistance in implementing a function to verify compliance regarding string formation, see Appendix C.2.

3.3. Eligible characters

Internationalized Domain Names for Applications [IDNA2008] defines a procedure in RFC 5892 [RFC5892] that determines code point sets allowed in domain names by calculating the value of a property for each code point, referred to as the Derived Property Value.

To define the eligible characters in Frogans address strings, the procedure for calculating the Derived Property Value is adapted by modifying the following Category Definitions (described in RFC 5892, section 2 Category Definitions Used to Calculate Derived Property Value), while leaving the algorithm (described in RFC 5892, section 3 Calculation of the Derived Property) unchanged:

* The Category Definition Exceptions (F) is modified by adding U+002A to the set of code points and by assigning the PVALID Derived Property Value to that code point. This modification reintroduces the U+002A ASTERISK character (the distinctive sign of a Frogans address) which is not allowed under the IDNA procedure.

  Furthermore, in order to ensure that the eligible characters in Frogans address strings are coherent with subsequent sections of this IFAP specification (Section 3.4, Section 4.4, and Section 5), an additional modification is applied to the Category Definition Exceptions (F). The following code points are added to the set of code points and are assigned the DISALLOWED Derived Property Value: U+02EC, U+A67F, U+A717, U+A718, U+A719, U+A71A, U+A71B, U+A71C, U+A71D, U+A71E, U+A71F, U+A788, U+A78D, and U+A7AA.

* The Category Definition Unstable (B) is modified so that it always returns False. In other words, no code points are unstable under this definition (characters that are not stable under NFKC are eliminated through the preceding rules stated in Section 3.2 of this IFAP specification). This modification reintroduces code points with the General Category of Lu (Uppercase_Letter), Lt (Titlecase_Letter), and Ll (Lowercase_Letter).

* The Category Definition LetterDigits (A) is modified by adding the General Category Lt (Titlecase_Letter) to the set of categories. This modification is necessary to ensure that code points with the General Category of Lt (Titlecase_Letter) are assigned the PVALID Derived Property Value.

Code points with the Derived Property Value of DISALLOWED or UNASSIGNED, as calculated following the adapted procedure described above, cannot be included in Frogans address strings.

The Unicode Technical Standard #39 [UTS39] defines a profile of identifiers in environments where security is an issue, referred to as General Security Profile for Identifiers (see the Unicode Technical Standard #39, section 3.1). This profile assigns either a Restricted or Allowed status to each character.

Code points with the Restricted status in Unicode Technical Standard #39 cannot be included in Frogans address strings, except for the U+002A ASTERISK character and characters belonging to the Unicode scripts defined as Aspirational Use Scripts or Limited Use Scripts in the Unicode Standard Annex #31.

Following the rules presented in Section 3.3, applied in addition to the preceding rules, a total of 93,929 code points and 60 Unicode scripts are available for use in Frogans address strings. Thus the rules presented in this section eliminate 11,261 code points and 42 Unicode scripts.

After having applied the preceding rules in this IFAP specification, the adapted procedure described above eliminates 4,619 code points that are allowed under the IDNA procedure. The adapted procedure also reintroduces 575 code points that are not allowed under the IDNA procedure: 510 code points with the General Category of Lu (Uppercase_Letter), 27 code points with the General Category of Lt (Titlecase_Letter), 37 code points with the General Category of Ll (Lowercase_Letter), and the U+002A ASTERISK character.

For assistance in implementing a function to verify compliance regarding eligible characters, see Appendix C.3.

3.4. Directionality

The Unicode Standard Annex #9 [UAX9] defines bidirectional character types (see the Unicode Standard Annex #9, section 3.2 Bidirectional Character Types) to manage text mixing both left-to-right and right-to-left writing directions. Each code point is assigned a bidirectional character type, referred to as its Bidi_Class property.

A total of 23 Bidi_Class property values are defined in the Unicode Standard Annex #9. After having applied the preceding rules in this IFAP specification, the code points in Frogans address strings can only have one of nine possible Bidi_Class property values. These property values are the following, with the total number of eligible characters for each one: L (Left-to-Right, 93,025), R (Right-to-Left,

102), AL (Right-to-Left Arabic, 279), EN (European Number, 20), ES (European Number Separator, 1), AN (Arabic Numbers, 10), NSM (Nonspacing Mark, 486), BN (Boundary Neutral, 2), or ON (Other Neutrals, 4).

These Bidi_Class property values fall into three main categories: Strong (which includes L, R, and AL), Weak (which includes EN, ES, AN, NSM, and BN), and Neutral (ON).

The following directionality rules apply to Frogans address strings:

* The Bidi_Class property value of the first code point of a Frogans address string equals either L, R, or AL. In other words, the first code point of a Frogans address belongs to the Strong category.

* If the Bidi_Class property value of the first code point of a Frogans address string equals L, then no other code point in the Frogans address string can have a Bidi_Class property value equal to R, AL, or AN. In addition, the Frogans address string ends with a code point with Bidi_Class property value L or EN, followed by zero or more code points with Bidi_Class property value NSM. As a result, in this case the directionality of the entire Frogans address string is left to right.

* If the Bidi_Class property value of the first code point of a Frogans address string equals R or AL, then no other code point in the Frogans address string can have a Bidi_Class property value equal to L. In addition, the Frogans address string ends with a code point with Bidi_Class property value R, AL, EN, or AN, followed by zero or more code points with Bidi_Class property value NSM. As a result, in this case the directionality of the entire Frogans address string is right to left.

Consequently, the first code point of Frogans address strings cannot have a Bidi_Class property value equal to EN, regardless of whether the directionality of the Frogans address string is right to left or left to right.

As a result of these rules, Frogans address strings cannot mix left-to-right and right-to-left directionality (except for code points having the EN or AN Bidi_Class property value in Frogans address strings with right-to-left directionality); and the Bidi_Class property of the first code point in a Frogans address string determines the directionality of the entire Frogans address string.

These directionality rules are intended to ensure that users reading a Frogans address string on screen or in print can easily and

unambiguously determine its directionality.

These rules are inspired by the Bidi Rule described in RFC 5893 [RFC5893], which is part of Internationalized Domain Names for Applications [IDNA2008] (see RFC 5893, section 2 The Bidi Rule).

The rules presented in Section 3.4, applied in addition to the preceding rules, do not reduce the total number of code points and Unicode scripts that are available for use in Frogans address strings.

For assistance in implementing a function to verify compliance regarding directionality, see Appendix C.4.

4. Structure of a Frogans address

The preceding section of this specification focuses on Frogans address strings, including the string character set, string formation, eligible characters, and directionality. This section describes the Frogans address structure.

The structure of Frogans addresses is the visible part of the iceberg in the definition of Frogans addresses. This structure provides Frogans addresses with a pattern that is easy to distinguish from other popular Internet address patterns such as those used in E-mail addresses or URLs.

4.1. Asterisk character

The structure of a Frogans address includes a special character that acts as a separator. This character, called the separator character, is the U+002A ASTERISK character (*).

A Frogans address contains one and only one separator character.

The separator character cannot be the first nor the last character of a Frogans address.

This separator character was chosen at the beginning of the Frogans project so as to avoid confusion with other separators such as the U+003A COLON character (:), the U+002F SOLIDUS character (/), and the U+002E FULL STOP character (.) that are commonly used in other computing environments.

The U+002A ASTERISK character in a Frogans address plays the same role as the U+0040 COMMERCIAL AT character (@) in an E-mail address, which separates the user from the host. The U+002A ASTERISK character separates the two parts of a Frogans address: the network name and the site name.

4.2. Network name

The network name of a Frogans address is used to represent the name of a Frogans network.

In a Frogans address, the network name is the string of characters that precedes the separator character when writing the Frogans address.

Thus in an address with left-to-right directionality, the network name is displayed to the left of the separator character. In a Frogans address with right-to-left directionality, the network name

is displayed to the right of the separator character.

Just like the entire Frogans address is an identifier (of a Frogans site), the network name alone is also an identifier (of a Frogans network). Certain restrictions apply to its first character.

The first character of the network name in a Frogans address cannot be:

* a combining character, i.e. a character with the General Category of M (Combining_Mark)

* a decimal number, i.e. a character with the General Category of Nd (Decimal_Number)

* any of the following characters: U+0375 GREEK LOWER NUMERAL SIGN, U+05F3 HEBREW PUNCTUATION GERESH, U+05F4 HEBREW PUNCTUATION GERSHAYIM, U+06FD SIGN SINDHI AMPERSAND, U+06FE ARABIC SIGN SINDHI POSTPOSITION MEN

4.3. Site name

The site name of a Frogans address is used to represent the name of a Frogans site within a Frogans network.

In a Frogans address, the site name is the string of characters that follows the separator character when writing the Frogans address.

Thus in an address with left-to-right directionality, the site name is displayed to the right of the separator character. In a Frogans address with right-to-left directionality, the site name is displayed to the left of the separator character.

Just like the entire Frogans address and the network name are identifiers, the site name alone is also an identifier (of a Frogans site within a Frogans network). Certain restrictions apply to its first character.

The first character of the site name of a Frogans address is subject to the same rules that apply to the first character of the network name of a Frogans address (see Section 4.2).

4.4. Connector characters

The structure of a Frogans addresses includes special characters that act as connectors. These characters, called connector characters, are the following:

- the U+002D HYPHEN-MINUS character
- the U+00B7 MIDDLE DOT character
- the U+30FB KATAKANA MIDDLE DOT character
- the U+0F0B TIBETAN MARK INTERSYLLABIC TSHEG character

The use of these connector characters is optional. One or more connector characters can be included in Frogans addresses to make it easier to read network names or site names that contain several words, by inserting connector characters between those words.

The following rules apply to the use of connector characters in the network name of a Frogans address:

* A connector character cannot be the first nor the last character of the network name.

* Two or more consecutive connector characters cannot be included in the network name.

* The character following a connector character in the network name cannot be a combining character. Combining characters are defined in Section 3.2.

The rules above concerning the use of connector characters in the network name of a Frogans address also apply to the site name of a Frogans address.

As a result of the rules defined in Section 3.3, the following characters are not eligible in Frogans addresses and therefore cannot be used to connect different words included in a network name or a site name: the U+0020 SPACE character (" "), the U+0027 APOSTROPHE character ('), the U+002E FULL STOP character ("."), and the U+003A COLON character (":").

For assistance in implementing a function to verify the structure of a Frogans address, see Appendix C.5.

5. Generating the reference form of a Frogans address

In order to generate the reference form of Frogans addresses, it is necessary to define and generate both the reference form of a network name and the reference form of a site name.

The reference form of a network name, a site name, or a Frogans address is generated from strings that comply with all the preceding rules in this specification.

The Unicode Standard [Unicode] defines a process to compare two identifiers for case-insensitive equality, referred to as caseless matching for identifiers. In this process, identifiers are compared by applying a string transformation and comparing the resulting strings. This string transformation is toNFKC_Casefold(NFD(X)), where X represents an identifier string (see the Unicode Standard, section 3.13 Default Case Algorithms, definition D147).

The reference form of the network name of a Frogans address is the string generated by applying to the network name the string transformation used in the process of caseless matching for identifiers defined in the Unicode Standard.

The reference form of the site name of a Frogans address is the string generated by applying to the site name the string transformation used in the process of caseless matching for identifiers defined in the Unicode Standard.

The reference form of a Frogans address is the string generated by concatenating the reference form of the network name, the separator character, and the reference form of the site name.

Since the separator character is not modified by the process of caseless matching for identifiers defined in the Unicode Standard, the reference form of a Frogans address is equivalent to the string generated by applying to the Frogans address the string transformation used in the process of caseless matching for identifiers defined in the Unicode Standard.

Due to the use of the caseless matching process, the number of code points in the reference form of a network name, a site name, or a Frogans address may be shorter or longer than the number of code points in that network name, site name, or Frogans address under certain conditions.

For example, the caseless matching process removes the U+200C ZERO WIDTH NON-JOINER code point. Conversely, it replaces the German lowercase character "Eszett" (U+00DF LATIN SMALL LETTER SHARP S) by

two code points (U+0073 LATIN SMALL LETTER S and U+0073 LATIN SMALL LETTER S).

The string transformation described in this section is coherent with the rules defined in previous sections of this specification. Thus the reference form also complies with all those rules.

The reference form of a network name, a site name, or a Frogans address is used to evaluate its length and to check whether two network names, site names, or Frogans addresses are identical. Unlike the preferred form of a network name, a site name, or a Frogans address, the reference form is not intended for display to end users.

For assistance in implementing a function to generate the reference form of a Frogans address, see Appendix C.6.

6. Evaluating the length of a Frogans address

In order to evaluate the length of Frogans addresses, it is necessary to define and evaluate both the length of a network name and the length of a site name.

The length of the network name of a Frogans address is the number of characters in the reference form of that network name.

The length of the site name of a Frogans address is the number of characters in the reference form of that site name.

The following rules apply to the length of the network name and site name in a Frogans address:

* The length of the network name is limited to between 1 and 28 characters.

* The length of the site name is limited to between 1 and 28 characters.

The length of a Frogans address equals the length of its network name plus one for the separator character plus the length of its site name. In other words, the length of a Frogans address equals the number of characters in the reference form of that Frogans address.

As a result of the preceding rules, the length of a Frogans address is limited to between 3 and 57 characters, including the network name, the separator character, and the site name.

7. Checking whether two Frogans addresses are identical

In order to check whether two Frogans addresses are identical, it is necessary to define both the rule used to check whether two network names are identical and the rule used to check whether two site names are identical.

Two network names are identical if the characters in their reference forms are the same.

Two site names are identical if the characters in their reference forms are the same.

Two Frogans addresses are identical if both their network names and site names are identical. In other words, two Frogans addresses are identical if the characters in their reference forms are the same.

For example, all the following network names are identical:

- mynetwork (reference form)
- MyNetwork
- MYNETWORK

For example, all the following site names are identical:

- mysite (reference form)
- MySite
- MYSITE

For example, all the following Frogans addresses are identical:

- mynetwork*mysite (reference form)
- MyNetwork*MYSITE
- MYNETWORK*MySite

However, the following Frogans addresses are not identical:

- my-network*MySite
- mynetwork*MySite

As a result of the method used to generate reference forms (see Section 5), two network names, site names, or Frogans addresses may be identical even though they do not have the same number of code points.

8. Usage of ASCII-encoded Frogans addresses

Unlike Internationalized Domain Names (IDNs) [IDNA2008], which are built upon ASCII-based domain names, Frogans addresses are based directly on the Unicode Standard [Unicode] and are international by design. Thus standard encoding methods such as UTF-8 [UTF-8] or UTF-16 [UTF-16] can generally be used for their transmission or storage.

However, UTF-8 and UTF-16, which produce binary sequences, may be unsuitable under certain specific circumstances such as:

* transmitting Frogans addresses using protocols requiring ASCII-encoded data

* using Frogans addresses in file names on file systems that do not support Unicode

Under such circumstances, it is necessary to encode Frogans addresses into ASCII [ASCII].

This section provides a uniform method for encoding Frogans addresses into ASCII to be used by applications that encounter these specific circumstances.

This method is simple: it uses 36 ASCII characters from 0 to 9 and from a to z (lowercase) and provides a fixed-length encoding scheme with four ASCII characters per code point. Given the maximum length of a Frogans address (see Section 6), the maximum number of characters in an ASCII-encoded Frogans address is 228.

ASCII-encoded Frogans addresses are used for technical purposes only. Except under the specific circumstances described above, ASCII-encoded Frogans addresses are not displayed to end users. For example, an application cannot use an ASCII-encoded Frogans address as a fall-back solution for displaying a Frogans address containing international characters that it cannot display correctly.

An ASCII-encoded Frogans address is generated using the following procedure.

First, the following three-step process is applied to each code point in the Frogans address string:

1. Given X the integer value of the code point, four integer values V1, V2, V3, and V4 are calculated as follows:

   V1 = ((X DIV 36) DIV 36) DIV 36
   V2 = ((X DIV 36) DIV 36) MOD 36
   V3 = (X DIV 36) MOD 36
   V4 = X MOD 36

   where DIV is an arithmetic operator which represents the integer division of one number by another, and MOD is an arithmetic operator which represents the remainder after an integer division of one number by another.

   As a result of the calculation, the values V2, V3 and V4 are between 0 and 35 inclusive. Since all code points defined in the Unicode Standard are lower than 1,114,111 (the code point U+10FFFF), the value V1 is between 0 and 23.

2. For each value Vi (where i ranges from 1 to 4), an ASCII character Ci is mapped as follows:

   * If the value Vi is between 0 and 9 inclusive, then the value of the ASCII code for character Ci equals (48+Vi), corresponding to the range of ASCII characters from 0 to 9.

   * If the value Vi is between 10 and 35 inclusive, then the value of the ASCII code for character Ci equals (87+Vi), corresponding to the range of lowercase ASCII characters from a to z.

3. A four-character ASCII string is generated by concatenating C1, C2, C3, and C4 in that order.

Examples:

* The four-character ASCII-encoded string representing the lowest code point for an eligible character in a Frogans address (the U+002A ASTERISK character) is 0016.

* The four-character ASCII-encoded string representing the highest code point for an eligible character in a Frogans address (the U+2B81D CJK UNIFIED IDEOGRAPH character) is 3ti5.

Second, after applying the above three-step process to each code point in the Frogans address string, all the generated four-character ASCII strings are concatenated in the order of the code points in the Frogans address string to create the ASCII-encoded Frogans address.

The uniform method provided above for encoding a Frogans address into ASCII also applies for encoding a network name or a site name into ASCII, should ASCII-encoded network names or site names be required in an application that encounters the specific circumstances described in the beginning of this section.

9. References

9.1. Normative references

[ASCII] American National Standards Institute (formerly United States of America Standards Institute), "USA Code for Information Interchange", ANSI X3.4-1968, 1968.

[RFC5892] Falstrom, P., "The Unicode Code Points and Internationalized Domain Names for Applications (IDNA)", RFC 5892, August 2010, <http://www.ietf.org/rfc/rfc5892.txt>.

[UAX9] The Unicode Consortium, "Unicode Standard Annex #9: Unicode Bidirectional Algorithm", Version 6.3.0, Revision 29, September 2013, <http://www.unicode.org/reports/tr9/tr9-29.html>.

[UAX15] The Unicode Consortium, "Unicode Standard Annex #15: Unicode Normalization Forms", Version Unicode 6.3.0, Revision 39, September 2013, <http://www.unicode.org/reports/tr15/tr15-39.html>.

[Unicode] The Unicode Consortium, "The Unicode Standard", Version 6.3.0, (Mountain View, CA: The Unicode Consortium, 2013. ISBN 978-1-936213-08-5), September 2013, <http://www.unicode.org/versions/Unicode6.3.0/>.

[UTS39] The Unicode Consortium, "Unicode Technical Standard #39: Unicode Security Mechanisms", Version 6.3.0, Revision 7, November 2013, <http://www.unicode.org/reports/tr39/tr39-7.html>.

9.2. Informative references

[BYLAWS] OP3FT, "Bylaws of the French Fonds de Dotation OP3FT, Organization for the Promotion, Protection and Progress of Frogans Technology", March 2012, <https://www.op3ft.org/en/resources/bylaws/access.html>.

[FNSL] STG Interactive S.A., "Frogans Network System Language", Version 3.0, May 2004, <https://www.frogans.org/en/resources/fnsl/access.html>.

This technical specification of the Frogans technology was granted free of charge and irrevocably by STG Interactive S.A. to the OP3FT, as part of the initial endowment of the OP3FT when the latter was created in 2012.

[IDNA2008]
Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, August 2010, <http://www.ietf.org/rfc/rfc5890.txt>.

IDNA2008 includes several additional documents: RFC 5891, RFC 5892, RFC 5893, RFC 5894, and RFC 5895.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, November 1987, <http://www.ietf.org/rfc/rfc1034.txt>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, <http://www.ietf.org/rfc/rfc2119.txt>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005, <http://www.ietf.org/rfc/rfc3986.txt>.

[RFC5893] Alvestrand, H. and C. Karp, "Right-to-Left Scripts for Internationalized Domain Names for Applications (IDNA)", RFC 5893, August 2010, <http://www.ietf.org/rfc/rfc5893.txt>.

[UAX31] The Unicode Consortium, "Unicode Standard Annex #31: Unicode Identifier and Pattern Syntax", Version 6.3.0, Revision 19, September 2013, <http://www.unicode.org/reports/tr31/tr31-19.html>.

[UTF-16] Hoffman, P. and F. Yergeau, "UTF-16, an encoding of ISO 10646", RFC 2781, February 2000, <http://www.ietf.org/rfc/rfc2781.txt>.

[UTF-8] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003, <http://www.ietf.org/rfc/rfc3629.txt>.

Appendix A. IFAP lookup tables

This appendix describes the IFAP lookup tables used in Appendix C which provides assistance in implementing this specification.

This appendix is not normative. Its contents do not replace the definitions and rules previously set forth in this specification, nor do they define any new rules.

IFAP lookup tables are files containing pre-processed lists of code points. This data is provided separately from this specification document in order to make the data easier to use for developers. IFAP lookup tables are accessible at the same permanent URL as this specification document (see the first page of this document).

Each IFAP lookup table is assigned a unique reference in LTnn_Label format, where nn is a zero-padded two-digit sequential number and Label is a label where words are separated by the underscore (_) character.

Each IFAP lookup table is provided in CSV format. The content of the file has the following characteristics:

* The file is encoded using the ASCII character set [ASCII]. Each line of the file ends with the ASCII character LF.

* The first lines in the file are comments starting with the ASCII character # (number sign). They include the IFAP lookup table reference, a brief description of its contents and use, the file name, and the file creation date. The comments also include: the list of third-party source materials and the list of other IFAP lookup tables used to create the lookup table; the description of the fields in the lookup table; and the method used to compute the field values in the lookup table.

* The first line of the file that is not a comment contains the field names of the lookup table, in uppercase, separated by the ASCII character comma (,).

* Each subsequent line of the file is a data line containing field values, separated by the ASCII character comma (,).

* The number of fields per data line remains constant. It is possible for a lookup table to contain only one field.

* The name of the first field is CODE_POINT. The value of this field represents either an individual code point or a continuous range of code points. Individual code points are represented in

'cphex' format, and ranges of code points in 'cphex1..cphex2' format, where 'cphex', 'cphex1', and 'cphex2' contain between four and six uppercase hexadecimal digits, and '..' is two consecutive ASCII full stop characters (.). The first and last points of a range are included in the range.

* The next fields contain information related to the code point or range of code points defined in the first field. Any code point included in the value of such fields is represented using the 'cphex' format described above. The value of such fields may be empty on some data lines.

* A code point cannot be listed in the first field of more than one data line, neither as an individual code point nor within a range. The data lines in the file are sorted in increasing order by the code point number of the first field.

* No comments are included between two data lines, at the end of a data line, or at the end of the file.

The remainder of this section lists all the 11 IFAP lookup tables used in Appendix C.

See the comments in each lookup table for a brief description of its contents and use.

The hash value provided for each IFAP lookup table is computed using the secure hash algorithm SHA-256 of the National Institute of Standards and Technology.

- Reference: LT01_Character_Set

  File name: ifap10-adopted.spec.lt01-character-set.txt
  File size: 10,833 bytes
  Total number of lines: 659
  Total number of data lines: 542
  File sha256 hash:
  1ffe10484791a3194e89a6b06a4165e3014d46db4b1350b714a5a54ca24bbea8

- Reference: LT02_Canonical_Mapping

  File name: ifap10-adopted.spec.lt02-canonical-mapping.txt
  File size: 228,876 bytes
  Total number of lines: 13,433
  Total number of data lines: 13,226
  File sha256 hash:
  29fd6a349286d97bfe49a4fd2adae0835cc98f4cf46420912d031e43fb725101

- Reference: LT03_Compatibility_Mapping

  File name: ifap10-adopted.spec.lt03-compatibility-mapping.txt
  File size: 52,480 bytes
  Total number of lines: 3,796
  Total number of data lines: 3,656
  File sha256 hash:
  ef488b52a974c07fd2c9926329ecf74d79bb3a731e7bf4f412d382a6dcacfc32

- Reference: LT04_Combining_Class

  File name: ifap10-adopted.spec.lt04-combining-class.txt
  File size: 8,685 bytes
  Total number of lines: 424
  Total number of data lines: 289
  File sha256 hash:
  c067080aa0f8014d274b38bbaef8f0c86ef6c43c69d199901d691d95e22c3953

- Reference: LT05_NFKC_Stable

  File name: ifap10-adopted.spec.lt05-nfkc-stable.txt
  File size: 9,846 bytes
  Total number of lines: 670
  Total number of data lines: 568
  File sha256 hash:
  8ff417be6fcb1b42eeb62025c220ddf64b038442a91b2e471f6060a48c90d1a4

- Reference: LT06_Combining_Marks

  File name: ifap10-adopted.spec.lt06-combining-marks.txt
  File size: 7,361 bytes
  Total number of lines: 339
  Total number of data lines: 216
  File sha256 hash:
  8e36bafab2858f93619db69eec3fe6a7e5c5f38cf1f51035b455e2c613d8d0a1

- Reference: LT07_Joining_Type

  File name: ifap10-adopted.spec.lt07-joining-type.txt
  File size: 8,628 bytes
  Total number of lines: 439
  Total number of data lines: 310
  File sha256 hash:
  97a52c54b6ba9239eebce4c81bdd37370d0b8ab6884de023cc86a4951c79d62c

- Reference: LT08_Eligible_Characters

File name: ifap10-adopted.spec.lt08-eligible-characters.txt
File size: 15,484 bytes
Total number of lines: 802
Total number of data lines: 588
File sha256 hash:
82837ce873df4afd5868d2b73b5b7b64e6f68dc403749d9d0d4736163c13c142

- Reference: LT09_Bidi_Class

File name: ifap10-adopted.spec.lt09-bidi-class.txt
File size: 7,953 bytes
Total number of lines: 329
Total number of data lines: 198
File sha256 hash:
ecf95ba655bc0d0f2826fab554af95e8a500ec626705806aad02ccdc4aa4999a

- Reference: LT10_Decimal_Numbers

File name: ifap10-adopted.spec.lt10-decimal-numbers.txt
File size: 5,383 bytes
Total number of lines: 156
Total number of data lines: 36
File sha256 hash:
c900eb69d2f90d7a93429e31d69099227de52937a209dab3e33b5fa02c7958c2

- Reference: LT11_NFKC_Case_Folding

File name: ifap10-adopted.spec.lt11-nfkc-case-folding.txt
File size: 11,743 bytes
Total number of lines: 714
Total number of data lines: 579
File sha256 hash:
d7cbfeebdf13b9c0b52f630a0b264300a4975e365280774f9eb7cdefdf370dd9

Appendix B. Pseudocode syntax

This appendix describes the syntax and conventions for the pseudocode used in Appendix C which provides assistance in implementing this specification.

This appendix is not normative. Its contents do not replace the definitions and rules previously set forth in this specification, nor do they define any new rules.

The pseudocode uses the following syntax and conventions.

All keywords are written in uppercase. The names of all functions, variables, and data objects are written in lowercase.

Spaces are used to separate elements.

Braces ({ and }) are used to delimit blocks of pseudocode.

To improve legibility, the text of the comments is not included in the pseudocode. Instead, comments are referenced by a number between angle brackets (< and >) at the end of a line. For example: <1> indicates comment number 1.

The following statements are used:

* FUNCTION: defines a function. The keyword FUNCTION is followed by the function name, then by a list of one or more parameter names between parentheses.

* VAR: defines a variable used in a function. The VAR keyword is followed by the name of the variable.

* RETURN: exits a function. They keyword RETURN is followed by the value returned by the function.

* CALL: calls a function. The keyword CALL is followed by the name of the called function, then by a list of one or more parameter values between parentheses. The list matches the definition of the called function.

* IF: tests an expression. The IF keyword is followed by the expression between parentheses, then by a block of pseudocode between braces to be executed if the expression evaluates to true.

* ELSE: follows an IF statement. The ELSE keyword is followed either by another IF statement or by a block of pseudocode, which are executed if the expression defined by the previous IF

statement evaluates to false. The pseudocode may contain cascading ELSE statements.

* FOR: defines a loop associated with an index. The FOR keyword is followed by the name of the index, the equal sign (=), the first value included in the index range, the TO keyword, then by the last value included in the index range, then by a block of pseudocode to be executed for each iteration of the loop. If the first or the last value of the index range is defined by an expression, then that expression is included between parentheses. If the last value in the index range is lower than the first value, then the TO keyword is replaced by the DOWNTO keyword. The index is incremented or decremented by one at each iteration of the loop.

* WHILE: defines a loop associated with an expression. The WHILE keyword is followed by the expression between parentheses, then by a block of pseudocode between braces to be executed for each iteration of the loop if the expression evaluates to true. Whenever the expression is evaluated to false, execution continues after the block of pseudocode.

* BREAK: exits a FOR or WHILE loop. The BREAK keyword is not followed by other keywords. Execution continues after the block of pseudocode defined in the loop.

The following logical expressions are used:

* (a == b) tests whether the value of a equals the value of b.

* (a != b) tests whether the value of a is different from the value of b.

* (c OR d) tests whether either of the expressions c or d evaluates to true.

* (c AND d) tests whether both the expressions c and d evaluate to true.

* (NOT c) negates the expression c.

Parentheses are used to combine groups of logical expressions.

The equal sign (=) is used in a block of pseudocode to assign a value to a variable.

The remainder of this section describes two data objects that are specific to the implementation of this specification:

* TABLE: defines a read-only data object containing an IFAP lookup table. For a description of the IFAP lookup table contents, see Appendix A.

* LIST: defines a read/write data object containing a list of code points.

The following methods are defined for a TABLE data object named my_table:

* my_table.CONTAINS (code_point): looks up in my_table a code point with the value of code_point. This method returns either true if a code point with value of code_point is found, or false otherwise.

* my_table.LOOKUP (code_point, field_name): looks up in my_table the value of the field called field_name for the code point equal to the value of code_point. When used in the pseudocode, the name of the field is preceded by the number sign (#). This method returns either the value of the field called field_name for the code point with the value of code_point, or NULL if there is no such code point.

* my_table.FIND (logical_expression): searches in my_table for a code point whose field values match certain conditions defined in the logical expression provided as a parameter. In the logical expression, the names of the fields that the conditions apply to are preceded by the number sign (#). This method returns either the value of a code point meeting the conditions, or NULL if there is no such code point.

The following property and methods are defined for a LIST data object named my_list:

* my_list.COUNT: returns the number of code points in the list

* my_list.GET (i): returns the value of the code point found at index i in the list. The range of index i is from 0 (the first code point) to (my_list.COUNT - 1) (the last code point in the list).

* my_list.APPEND (code_point_series): appends one or more code points to the list. The code points to append are provided as arguments separated by commas.

* my_list.SET (i, code_point): sets the code point found at index i in the list to the value of code_point.

* my_list.REMOVE (i): removes the code point at index i from the list.

Appendix C. Assistance in implementing the specification

This appendix provides a series of processes that can be used to implement this specification.

This appendix is not normative. Its contents do not replace the definitions and rules previously set forth in this specification, nor do they define any new rules.

This appendix does not cover the following parts of the specification, as they do not present any particular implementation difficulties: Evaluating the length of a Frogans address (Section 6), Checking whether two Frogans addresses are identical (Section 7), and Usage of ASCII-encoded Frogans addresses (Section 8),

Given the limited length of Frogans addresses (see Section 6), the processes are designed to minimize the size of the IFAP lookup tables rather than to optimize process performance.

The six sections in this appendix provide for each function: the function name and description; the functions it is called by and the functions it calls; the IFAP lookup tables used by the function; the input parameters; the possible values returned by the function; a numbered list of comments related to the pseudocode; and finally pseudocode describing the function. Comments in the pseudocode are indicated by a number between angle brackets (< and >).

C.1. String character set

This section provides assistance in implementing a process that verifies whether the code points of a candidate string are in the character set applicable to Frogans addresses.

One function is required to implement this process:

FUNCTION |c1_verify_character_set|

Description:
This is the main function for this process.

It verifies each code point in the candidate string by performing a look-up in IFAP lookup table LT01_Character_Set. If any code point in the candidate string is not found, then it is invalid and the entire candidate string is rejected. Otherwise, if all the code point look-ups are successful, then the candidate string is accepted.

Called by:
none

Calls:
none

IFAP lookup tables used:
- LT01_Character_Set

Input:
- codepoints: a LIST data object containing code points that represent the candidate string

Returns:
true if the candidate string is accepted, or false otherwise

Comments:
none

Pseudocode:

```
,-------------------------------------------------------------.
| FUNCTION c1_verify_character_set (codepoints)               |
| {                                                           |
|   TABLE table_LT01                                          |
|   VAR cur_cp                                                |
|   VAR index                                                 |
|   FOR index = 0 TO (codepoints.COUNT - 1)                   |
|   {                                                         |
|     cur_cp = codepoints.GET (index)                         |
|     IF (NOT table_LT01.CONTAINS (cur_cp))                   |
|     {                                                       |
|       RETURN false                                          |
|     }                                                       |
|   }                                                         |
|   RETURN true                                               |
| }                                                           |
`-------------------------------------------------------------'
```

C.2. String formation

This section provides assistance in implementing a process that verifies whether a candidate string is compliant with string formation.

Eight functions are required to implement this process:

FUNCTION |c2_verify_string_formation|

Description:
This is the main function for this process.

It generates an NFKC normalized string from the candidate string and then compares the two strings. If there is any difference whatsoever, i.e. if their code points are not exactly the same or are not in exactly the same order, then the candidate string is rejected.

Otherwise, the function checks whether the candidate string contains more than 30 consecutive combining characters. This involves looking up each code point in the candidate string in IFAP lookup table LT06_Combining_Marks to determine whether the code point is a combining character. If the candidate string contains more than 30 consecutive combining characters, then the candidate string is rejected.

Otherwise, the function checks whether the candidate string contains either the U+200C ZERO WIDTH NON-JOINER or the U+200D ZERO WIDTH JOINER code point. If so, it checks whether those code points meet the contextual conditions for being included in a Frogans address string. If the code point does not meet those conditions, then the candidate string is rejected.

Otherwise the candidate string is accepted.

Called by:
none

Calls:
- |c2_normalize_nfkc|
- |c2_verify_joiner_200c_sequence|
- |c2_verify_joiner_virama|

IFAP lookup tables used:
- LT06_Combining_Marks

Input:
- codepoints: a LIST data object containing code points that represent the candidate string

Returns:
true if the candidate string is accepted, or false otherwise

Comments:
none

Pseudocode:

```
,-------------------------------------------------------------.
| FUNCTION c2_verify_string_formation (codepoints)            |
| {                                                           |
|   TABLE table_LT06                                          |
|   LIST work_cps                                             |
|   VAR index                                                 |
|   VAR cur_cp                                                |
|   VAR combining_marks_count                                 |
|   work_cps = CALL c2_normalize_nfkc (codepoints)            |
|   IF (work_cps != codepoints)                               |
|   {                                                         |
|     RETURN false                                            |
|   }                                                         |
|   combining_marks_count = 0                                 |
|   FOR index = 0 TO (codepoints.COUNT - 1)                   |
|   {                                                         |
|     cur_cp = codepoints.GET (index)                         |
|     IF (table_LT06.CONTAINS (cur_cp))                       |
|     {                                                       |
|       combining_marks_count = combining_marks_count + 1     |
|       IF (combining_marks_count > 30)                       |
|       {                                                     |
|         RETURN false                                        |
|       }                                                     |
|     }                                                       |
|     ELSE                                                    |
|     {                                                       |
|       combining_marks_count = 0                             |
|     }                                                       |
|   }                                                         |
|   FOR index = 0 TO (codepoints.COUNT - 1)                   |
|   {                                                         |
|     cur_cp = codepoints.GET (index)                         |
|     IF (cur_cp == U+200C)                                   |
|     {                                                       |
|       IF (CALL c2_verify_joiner_200c_sequence               |
|                                (codepoints, index) == false)|
|       {                                                     |
|         IF (CALL c2_verify_joiner_virama                    |
|                                (codepoints, index) == false)|
|         {                                                   |
|           RETURN false                                      |
|         }                                                   |
|       }                                                     |
|     }                                                       |
|     IF (cur_cp == U+200D)                                   |
```

```
|     {                                                          |
|       IF (CALL c2_verify_joiner_virama                         |
|                                (codepoints, index) == false) |
|       {                                                        |
|         RETURN false                                           |
|       }                                                        |
|     }                                                          |
|   }                                                            |
|   RETURN true                                                  |
| }                                                              |
`----------------------------------------------------------------'
```

FUNCTION |c2_normalize_nfkc|

Description:
This is a sub-function of the string formation process.

It applies a three-step procedure to generate an NFKC normalized string from an input string of code points.

Called by:
- |c2_verify_string_formation|

Calls:
- |c2_decompose_compatibility|
- |c2_reorder|
- |c2_compose|

IFAP lookup tables used:
- none

Input:
- codepoints: a LIST data object containing code points that represent the string to be normalized

Returns:
the NFKC normalized string

Comments:
none

Pseudocode:

```
,----------------------------------------------------------------.
| FUNCTION c2_normalize_nfkc (codepoints)                        |
| {                                                              |
|   LIST work_cps                                                |
|   work_cps = codepoints                                        |
```

```
|   work_cps = CALL c2_decompose_compatibility (work_cps)     |
|   work_cps = CALL c2_reorder (work_cps)                     |
|   work_cps = CALL c2_compose (work_cps)                     |
|   RETURN work_cps                                           |
| }                                                           |
`-------------------------------------------------------------'
```

FUNCTION |c2_decompose_compatibility|

Description:
This is a sub-function of the string formation process.

It is part of step 1 in the three-step procedure for generating an NFKC normalized string from an input string of code points.

This function performs a compatibility decomposition on each code point in the input string.

Called by:
- |c2_normalize_nfkc|

Calls:
- |c2_decompose_compatibility_cp|

IFAP lookup tables used:
- none

Input:
- codepoints: a LIST data object containing code points that represent the string to be decomposed

Returns:
a string containing the compatibility decomposition of each code point in the input string

Comments:
none

Pseudocode:

```
,-------------------------------------------------------------.
| FUNCTION c2_decompose_compatibility (codepoints)            |
| {                                                           |
|   LIST work_cps                                             |
|   LIST temporary_cps                                        |
|   VAR cur_cp                                                |
|   VAR index                                                 |
|   FOR index = 0 TO (codepoints.COUNT - 1)                   |
```

```
|   {                                                          |
|     cur_cp = codepoints.GET (index)                          |
|     temporary_cps = CALL c2_decompose_compatibility_cp       |
|                                                  (cur_cp)    |
|     work_cps.APPEND (temporary_cps)                          |
|   }                                                          |
|   RETURN work_cps                                            |
| }                                                            |
`--------------------------------------------------------------'
```

FUNCTION |c2_decompose_compatibility_cp|

Description:

This is a sub-function of the string formation process.

It is part of step 1 in the three-step procedure for generating an NFKC normalized string from an input string of code points.

This function uses a recursive algorithm to decompose a code point. This requires examining the canonical decomposition of the input code point, first in IFAP lookup table LT02_Canonical_Mapping and then in IFAP lookup table LT03_Compatibility_Mapping. A given code point cannot exist in both tables. If a code point does not exist in either table, then it is included in the normalized string as it is.

The recursive algorithm in this function is based on the rules set forth in the Unicode Standard [Unicode] section 3.7 Decomposition, D65 compatibility decomposition.

Called by:

- |c2_decompose_compatibility|
- |c2_decompose_compatibility_cp|. The function calls itself recursively.

Calls:

- |c2_decompose_compatibility_cp|. The function calls itself recursively.

IFAP lookup tables used:

- LT02_Canonical_Mapping
- LT03_Compatibility_Mapping

Input:

- a_codepoint: the code point to be decomposed

Returns:
a list of code points representing the decomposed form of the input code point

Comments:
<1> if cur_cp exists in the table, the function calls itself
<2> if cur_cp exists in the table, the function calls itself

Pseudocode:

```
FUNCTION c2_decompose_compatibility_cp (a_codepoint)
{
  TABLE table_LT02
  TABLE table_LT03
  LIST decomposition_cps
  LIST work_cps
  VAR cur_cp
  VAR index
  IF (table_LT02.CONTAINS (a_codepoint))
  {
    decomposition_cps = table_LT02.LOOKUP (a_codepoint,
                                      #canonical_mapping)
    FOR index = 0 TO (decomposition_cps.COUNT - 1)
    {
      cur_cp = decomposition_cps.GET (index)
      work_cps.APPEND (CALL c2_decompose_compatibility_cp
                                              (cur_cp))  <1>
    }
    RETURN work_cps
  }
  IF (table_LT03.CONTAINS (a_codepoint))
  {
    decomposition_cps = table_LT03.LOOKUP (a_codepoint,
                                   #compatibility_mapping)
    FOR index = 0 TO (decomposition_cps.COUNT - 1)
    {
      cur_cp = decomposition_cps.GET (index)
      work_cps.APPEND (CALL c2_decompose_compatibility_cp
                                              (cur_cp))  <2>
    }
    RETURN work_cps
  }
  work_cps.APPEND (a_codepoint)
  RETURN work_cps
}
```

FUNCTION |c2_reorder|

Description:
This is a sub-function used in both the string formation process and in the process for generating the reference form.

It is part of step 2 in the procedure for generating a normalized string from an input string of code points. It is called by three different functions: one to generate the NFKC form, the second to generate the NFD form, and the third to generate the NFC form.

After the code points have been decomposed in Step 1, they are reordered according to the rules set forth in the Unicode Standard, Section 3.11 Normalization Form, D109 Canonical Ordering Algorithm. This requires examining the combining class of each code point in IFAP lookup table LT04_Combining_Class.

Called by:
- |c2_normalize_nfkc|
- |c6_normalize_nfd|
- |c6_normalize_nfc|

Calls:
none

IFAP lookup tables used:
- LT04_Combining_Class

Input:
- codepoints: a LIST data object containing code points that represent the string to be reordered

Returns:
a list of code points representing the reordered string

Comments:
<1> Examine and compare the canonical combining class of the previous and the current code point in the input string.

Pseudocode:

```
,-------------------------------------------------------------.
| FUNCTION c2_reorder (codepoints)                            |
| {                                                           |
|   TABLE table_LT04                                          |
|   LIST work_cps                                             |
```

```
|   VAR swapped                                                  |
|   VAR index                                                    |
|   VAR prev_ccc                                                 |
|   VAR cur_ccc                                                  |
|   VAR temp_cp                                                  |
|   work_cps = codepoints                                        |
|   swapped = true                                               |
|   WHILE (swapped)                                              |
|   {                                                            |
|     swapped = false                                            |
|     FOR index = 1 TO (work_cps.COUNT - 1)                      |
|     {                                                   <1>    |
|       prev_ccc = table_LT04.LOOKUP                             |
|                  (work_cps.GET (index - 1),                    |
|                      #canonical_combining_class)               |
|       IF (prev_ccc == NULL)                                    |
|       {                                                        |
|         prev_ccc = 0                                           |
|       }                                                        |
|       cur_ccc = table_LT04.LOOKUP                              |
|                  (work_cps.GET (index),                        |
|                      #canonical_combining_class)               |
|       IF (cur_ccc == NULL)                                     |
|       {                                                        |
|         cur_ccc = 0                                            |
|       }                                                        |
|       IF ((cur_ccc != 0) AND                                   |
|           (prev_ccc > 0) AND                                   |
|           (prev_ccc > cur_ccc)                                 |
|          )                                                     |
|       {                                                        |
|         temp_cp = work_cps.GET (index - 1)                     |
|         work_cps.SET (index - 1, work_cps.GET (index))         |
|         work_cps.SET (index, temp_cp)                          |
|         swapped = true                                         |
|       }                                                        |
|     }                                                          |
|   }                                                            |
|   RETURN work_cps                                              |
| }                                                              |
`----------------------------------------------------------------'
```

FUNCTION |c2_compose|

Description:
This is a sub-function used in both the string formation process and in the process for generating the reference form.

It is part of step 3 in the procedure for generating a normalized string from a candidate string of code points. It is called by two different functions: one to generate the NFKC form and the second to generate the NFC form.

After all the code points have been decomposed in Step 1 and then reordered in Step 2, they are re-composed to create the normalization form required. The composition procedure is based on the rules set forth in the Unicode Standard section 3.11 Normalization Forms, D117 Canonical Composition Algorithm.

The function examines all the code points in the input string to determine whether it contains two code points that can be combined, depending on their canonical combining class (ccc) read in IFAP lookup table LT04_Combining_Class. If so, it combines those code points into a single code point. Then it continues to examine the rest of the input string.

Called by:
- |c2_normalize_nfkc|
- |c6_normalize_nfd|
- |c6_normalize_nfc|

Calls:
none

IFAP lookup tables used:
- LT02_Canonical_Mapping
- LT04_Combining_Class

Input:
- codepoints: a LIST data object containing code points that represent the string to be composed

Returns:
a list of code points representing the composed input string.

Comments:
<1> Starter code point in the code point string. For a code point to be a valid starter, the value of the CANONICAL_COMBINING_CLASS field (ccc) in IFAP lookup table LT04_Combining_Class must equal 0.
<2> For each code point in temporary_cps, determine its starter, previous, and current code points.

<3> Also determine the ccc of the next and previous code points.
<4> Read each line in IFAP lookup table LT02_Canonical_Mapping to determine whether starter_cp and cur_cp can be combined into a single code point. If so, set the composite variable to the combined code point
<5> If these conditions are met, then replace code point at starter_index with the value of the composite variable and remove the temporary code point used for the composition.
<6> If true, then the code point at index is a valid starter code point.

Pseudocode:

```
,-------------------------------------------------------------.
| FUNCTION c2_compose (codepoints)                            |
| {                                                           |
|   TABLE table_LT02                                          |
|   TABLE table_LT04                                          |
|   LIST temporary_cps                                        |
|   VAR starter_index                                         |
|   VAR starter_ccc                                           |
|   VAR starter_cp                                      <1>   |
|   VAR prev_cp                                               |
|   VAR cur_cp                                                |
|   VAR prev_ccc                                              |
|   VAR cur_ccc                                               |
|   VAR composite                                             |
|   LIST candidate_cps                                        |
|   VAR index                                                 |
|   VAR length                                                |
|   temporary_cps = codepoints                                |
|   starter_index = 0                                         |
|   starter_cp = temporary_cps.GET (starter_index)            |
|   starter_ccc = table_LT04.LOOKUP                           |
|                (starter_cp, #canonical_combining_class)     |
|   IF (starter_ccc == NULL)                                  |
|   {                                                         |
|     starter_ccc = 0                                         |
|   }                                                         |
|   length = temporary_cps.COUNT                              |
|   index = 1                                                 |
|   WHILE (index < length)                                    |
|   {                                                   <2>   |
|     starter_cp = temporary_cps.GET (starter_index)          |
|     prev_cp = temporary_cps.GET (index - 1)                 |
|     cur_cp = temporary_cps.GET (index)                      |
|     prev_ccc = table_LT04.LOOKUP                      <3>   |
|                   (prev_cp, #canonical_combining_class)     |
```

```
|     IF (prev_ccc == NULL)                                          |
|     {                                                              |
|       prev_ccc = 0                                                 |
|     }                                                              |
|     cur_ccc = table_LT04.LOOKUP                                    |
|                    (cur_cp, #canonical_combining_class)            |
|     IF (cur_ccc == NULL)                                           |
|     {                                                              |
|       cur_ccc = 0                                                  |
|     }                                                              |
|     composite = 0                                                  |
|     IF (starter_ccc == 0)                                          |
|     {                                                              |
|       candidate_cps.APPEND (starter_cp, cur_cp)                    |
|       composite = table_LT02.FIND (                          <4>  |
|                     (#full_composition_exclusion == 0)             |
|                        AND (#canonical_mapping ==                  |
|                                          candidate_cps)            |
|       IF (composite == NULL)                                       |
|       {                                                            |
|         composite = 0                                              |
|       }                                                            |
|     }                                                              |
|     IF ((composite != 0) AND                                       |
|         ((prev_ccc < cur_ccc) OR (prev_ccc == 0))             <5>  |
|        )                                                           |
|     {                                                              |
|       temporary_cps.SET (starter_index, composite)                 |
|       temporary_cps.REMOVE (index)                                 |
|       length = length - 1                                          |
|     }                                                              |
|     ELSE                                                           |
|     {                                                              |
|       IF (cur_ccc == 0)                                       <6>  |
|       {                                                            |
|         starter_index = index                                      |
|         starter_ccc = 0                                            |
|       }                                                            |
|       temporary_cps.SET (index, cur_cp)                            |
|       index = index + 1                                            |
|     }                                                              |
|   }                                                                |
|   RETURN temporary_cps                                             |
| }                                                                  |
`--------------------------------------------------------------------'
```

FUNCTION |c2_verify_joiner_200c_sequence|

Description:
This is a sub-function of the string formation process.

This function checks whether the U+200C ZERO WIDTH NON-JOINER code point at the joiner_index position in the candidate string meets the condition defined in Section 3.2 related to the Joining_Type property.

It searches for the required pattern before the U+200C ZERO WIDTH NON-JOINER code point, and then searches for the required pattern after that code point. If one of the two required patterns is not found, then the candidate string is rejected. Otherwise the candidate string is accepted.

Called by:
- |c2_verify_string_formation|

Calls:
- none

IFAP lookup tables used:
- LT07_Joining_Type

Input:
- codepoints: a LIST data object containing code points that represent the candidate string
- joiner_index: an index indicating the position of the U+200C ZERO WIDTH NON-JOINER code point in the candidate string

Returns:
true if the candidate string is accepted, or false otherwise

Comments:
<1> search for a valid starting sequence before the U+200C ZERO WIDTH NON-JOINER code point in the candidate string.
<2> search for a valid ending sequence after the U+200C ZERO WIDTH NON-JOINER code point in the candidate string.

Pseudocode:

```
,-------------------------------------------------------------.
| FUNCTION c2_verify_joiner_200c_sequence (codepoints,        |
|                                          joiner_index)      |
| {                                                           |
|   TABLE table_LT07                                          |
|   VAR index                                                 |
|   VAR joining_type                                          |
|   VAR start_found                                           |
```

```
|   VAR end_found                                                    |
|   VAR cur_cp                                                       |
|   IF ((codepoints.COUNT < 3) OR (joiner_index == 0) OR             |
|       (joiner_index == codepoints.COUNT - 1)                       |
|      )                                                             |
|   {                                                                |
|      RETURN false                                                  |
|   }                                                                |
|   start_found = false                                        <1>   |
|   FOR index = (joiner_index - 1) DOWNTO 0                          |
|   {                                                                |
|     joining_type = table_LT07.LOOKUP                               |
|                      (codepoints.GET (index), #joining_type)       |
|     IF (joining_type == NULL)                                      |
|     {                                                              |
|        joining_type = 'U'                                          |
|     }                                                              |
|     IF (joining_type != 'T')                                       |
|     {                                                              |
|       IF ((joining_type == 'D') OR                                 |
|           (joining_type == 'L'))                                   |
|       {                                                            |
|         start_found = true                                         |
|       }                                                            |
|       BREAK                                                        |
|     }                                                              |
|   }                                                                |
|   IF (NOT start_found)                                             |
|   {                                                                |
|      RETURN false                                                  |
|   }                                                                |
|   end_found = false                                          <2>   |
|   FOR index = (joiner_index + 1) TO (codepoints.COUNT - 1)         |
|   {                                                                |
|     joining_type = table_LT07.LOOKUP                               |
|                      (codepoints.GET (index), #joining_type)       |
|     IF (joining_type == NULL)                                      |
|     {                                                              |
|        joining_type = 'U'                                          |
|     }                                                              |
|     IF (joining_type != 'T')                                       |
|     {                                                              |
|       IF ((joining_type == 'D') OR                                 |
|           (joining_type == 'R'))                                   |
|       {                                                            |
|         end_found = true                                           |
|       }                                                            |
|       BREAK                                                        |
```

```
|     }                                                          |
|    }                                                           |
|   IF (NOT end_found)                                           |
|   {                                                            |
|     RETURN false                                               |
|   }                                                            |
|   RETURN true                                                  |
| }                                                              |
`----------------------------------------------------------------'
```

FUNCTION |c2_verify_joiner_virama|

Description:

This is a sub-function of the string formation process.

This function checks whether the code point at the joiner_index position in the candidate string is preceded by a code point with the Canonical_Combining_Class property value equal to 9 (Virama) as described in Section 3.2. If not, the candidate string is rejected. Otherwise the candidate string is accepted.

This function can be used for candidate strings with either the U+200C ZERO WIDTH NON-JOINER code point or the U+200D ZERO WIDTH JOINER code point.

Called by:

- |c2_verify_string_formation|

Calls:

- none

IFAP lookup tables used:

- LT04_Combining_Class

Input:

- codepoints: a LIST data object containing code points that represent the candidate string
- joiner_index: an index indicating the position of either the U+200C ZERO WIDTH NON-JOINER or U+200D ZERO WIDTH JOINER code point in the candidate string

Returns:

true if the candidate string is accepted, or false otherwise

Comments:

none

Pseudocode:

```
FUNCTION c2_verify_joiner_virama (codepoints, joiner_index)
{
  TABLE table_LT04
  VAR previous_cp
  VAR ccc
  IF (joiner_index == 0)
  {
     RETURN false
  }
  previous_cp = codepoints.GET (joiner_index - 1)
  ccc = table_LT04.LOOKUP
           (previous_cp, #canonical_combining_class)
  IF (ccc == NULL)
  {
     ccc = 0
  }
  IF (ccc != 9)
  {
     RETURN false
  }
  RETURN true
}
```

C.3. Eligible characters

This section provides assistance in implementing a process that verifies whether a candidate string contains only eligible characters.

One function is required to implement this process:

FUNCTION |c3_verify_eligible_characters|

Description:
This is the main function for the process to determine whether characters are eligible.

It verifies whether each code point in the candidate string is eligible to be used in a Frogans address. If any of the code points are not eligible, then the entire candidate string is rejected. Otherwise the candidate string is accepted.

Called by:
- none

Calls:
- none

IFAP lookup tables used:
- LT08_Eligible_Characters

Input:
- codepoints: a LIST data object containing code points that represent the candidate string

Returns:
true if the candidate string is accepted, or false otherwise

Comments:
none

Pseudocode:

```
,-------------------------------------------------------------.
| FUNCTION c3_verify_eligible_characters (codepoints)         |
| {                                                           |
|   TABLE table_LT08                                          |
|   VAR index                                                 |
|   VAR cur_cp                                                |
|   VAR eligibility                                           |
|   FOR index = 0 TO (codepoints.COUNT - 1)                   |
|   {                                                         |
|     cur_cp = codepoints.GET (index)                         |
|     eligibility = table_LT08.LOOKUP (cur_cp, #is_eligible)  |
|     IF ((eligibility == NULL) OR                            |
|         (eligibility == false))                             |
|     {                                                       |
|       RETURN false                                          |
|     }                                                       |
|   }                                                         |
|   RETURN true                                               |
| }                                                           |
`-------------------------------------------------------------'
```

C.4. Directionality

This section provides assistance in implementing a process that verifies whether a candidate string complies with directionality rules.

The functions described in this section are designed to verify the directionality of an entire Frogans address. These functions can be easily modified to verify the directionality of a network name or a site name. For network names, the modifications involve removing the directionality rule applicable to the end of the string. For site names, the modifications involve adding a parameter to provide the directionality of the associated network name, and removing the directionality rule applicable to the first character of the string.

Three functions are required to implement this process:

FUNCTION |c4_verify_directionality|

Description:
This is the main function for the directionality process.

It verifies that the candidate string follows the directionality rules for Frogans address strings.

First the function looks up the first code point in the candidate string to determine its directionality. Then, depending on the directionality of the first code point, it calls either the |c4_verify_ltr| or the |c4_verify_rtl| function to verify the directionality of the candidate string. If any of the code points in the candidate string do not comply with the IFAP directionality rules, then the entire candidate string is rejected. Otherwise the candidate string is accepted.

Called by:
- none

Calls:
- |c4_verify_ltr|
- |c4_verify_rtl|

IFAP lookup tables used:
- LT09_Bidi_Class

Input:
- codepoints: a LIST data object containing code points that represent the candidate string

Returns:
true if the candidate string is accepted, or false otherwise.

Comments:

<1> returns false because the first code point does not have a strong directionality

Pseudocode:

```
,-------------------------------------------------------------.
| FUNCTION c4_verify_directionality (codepoints)              |
| {                                                           |
|   TABLE table_LT09                                          |
|   VAR first_cp                                              |
|   VAR bidi_class                                            |
|   first_cp = codepoints.GET (0)                             |
|   bidi_class = table_LT09.LOOKUP (first_cp, #bidi_class)    |
|   IF (bidi_class == NULL)                                   |
|   {                                                         |
|     bidi_class = 'L'                                        |
|   }                                                         |
|   IF (bidi_class == 'L')                                    |
|   {                                                         |
|     if (CALL c4_verify_ltr (codepoints) == false)           |
|     {                                                       |
|        RETURN false                                         |
|     }                                                       |
|   }                                                         |
|   ELSE IF ((bidi_class == 'R') OR                           |
|            (bidi_class == 'AL'))                            |
|   {                                                         |
|     if (CALL c4_verify_rtl (codepoints) == false)           |
|     {                                                       |
|        RETURN false                                         |
|     }                                                       |
|   }                                                         |
|   ELSE                                                      |
|   {                                                         |
|     RETURN false                                    <1>     |
|   }                                                         |
|  RETURN true                                                |
| }                                                           |
`-------------------------------------------------------------'
```

FUNCTION |c4_verify_ltr|

Description:
This is a sub-function of the directionality process.

It verifies whether the candidate string complies with the directionality rules concerning left-to-right Frogans address strings. First it checks that all the code points in the

candidate string have a bidi_class that is compatible with a left-to-right Frogans address. If any of the code points in the candidate string do not comply, then the entire candidate string is rejected.

Otherwise, it checks that the end of the Frogans address is compatible with a left-to-right Frogans address. If this is not the case, then the candidate string is rejected.

Otherwise, the candidate string is accepted.

Called by:
- |c4_verify_directionality|

Calls:
none

IFAP lookup tables used:
- LT09_Bidi_Class

Input:
- codepoints: a LIST data object containing code points that represent a candidate string.

Returns:
true if the candidate string is accepted, or false otherwise.

Comments:
none

Pseudocode:

```
,-------------------------------------------------------------.
| FUNCTION c4_verify_ltr (codepoints)                         |
| {                                                           |
|   TABLE table_LT09                                          |
|   VAR index                                                 |
|   VAR cur_cp                                                |
|   VAR bidi_class                                            |
|   FOR index = 1 TO (codepoints.COUNT - 1)                   |
|   {                                                         |
|     cur_cp = codepoints.GET (index)                         |
|     bidi_class = table_LT09.LOOKUP (cur_cp, #bidi_class)    |
|     IF (bidi_class == NULL)                                 |
|     {                                                       |
|       bidi_class = 'L'                                      |
|     }                                                       |
|     IF ((bidi_class == 'R') OR                              |
```

```
|         (bidi_class == 'AL') OR                                  |
|         (bidi_class == 'AN'))                                    |
|     {                                                            |
|       RETURN false                                               |
|     }                                                            |
|   }                                                              |
|   FOR index = (codepoints.COUNT - 1) DOWNTO 1                    |
|   {                                                              |
|     cur_cp = codepoints.GET (index)                              |
|     bidi_class = table_LT09.LOOKUP (cur_cp, #bidi_class)         |
|     IF (bidi_class == NULL)                                      |
|     {                                                            |
|       bidi_class = 'L'                                           |
|     }                                                            |
|     IF (bidi_class != 'NSM')                                     |
|     {                                                            |
|       IF ((bidi_class != 'L') AND                                |
|           (bidi_class != 'EN'))                                  |
|       {                                                          |
|         RETURN false                                             |
|       }                                                          |
|       BREAK                                                      |
|     }                                                            |
|   }                                                              |
|   RETURN true                                                    |
| }                                                                |
`------------------------------------------------------------------'
```

FUNCTION |c4_verify_rtl|

Description:
This is a sub-function of the directionality process.

It verifies whether the candidate string complies with the directionality rules concerning right-to-left Frogans address strings. First it checks that all the code points in the candidate string have a bidi_class that is compatible with a right-to-left Frogans address. If any of the code points in the candidate string do not comply, then the entire candidate string is rejected.

Otherwise, it checks that the end of the Frogans address is compatible with a right-to-left Frogans address. If this is not the case, then the candidate string is rejected.

Otherwise, the candidate string is accepted.

Called by:
- |c4_verify_directionality|

Calls:
none

IFAP lookup tables used:
- LT09_Bidi_Class

Input:
- codepoints: a LIST data object containing code points that represent a candidate string.

Returns:
true if the candidate string is accepted, or false otherwise.

Comments:
none

Pseudocode:

```
,-------------------------------------------------------------.
| FUNCTION c4_verify_rtl (codepoints)                         |
| {                                                           |
|   TABLE table_LT09                                          |
|   VAR index                                                 |
|   VAR cur_cp                                                |
|   VAR bidi_class                                            |
|   FOR index = 1 TO (codepoints.COUNT - 1)                   |
|   {                                                         |
|     cur_cp = codepoints.GET (index)                         |
|     bidi_class = table_LT09.LOOKUP (cur_cp, #bidi_class)    |
|     IF (bidi_class == NULL)                                 |
|     {                                                       |
|       bidi_class = 'L'                                      |
|     }                                                       |
|     IF (bidi_class == 'L')                                  |
|     {                                                       |
|       RETURN false                                          |
|     }                                                       |
|   }                                                         |
|   FOR index = (codepoints.COUNT - 1) DOWNTO 1               |
|   {                                                         |
|     cur_cp = codepoints.GET (index)                         |
|     bidi_class = table_LT09.LOOKUP (cur_cp, #bidi_class)    |
|     IF (bidi_class == NULL)                                 |
|     {                                                       |
|       bidi_class = 'L'                                      |
```

```
|     }                                                          |
|     IF (bidi_class != 'NSM')                                   |
|     {                                                          |
|       IF ((bidi_class != 'R') AND                              |
|           (bidi_class != 'AL') AND                             |
|           (bidi_class != 'EN') AND                             |
|           (bidi_class != 'AN'))                                |
|       {                                                        |
|         RETURN false                                           |
|       }                                                        |
|       BREAK                                                    |
|     }                                                          |
|   }                                                            |
|   RETURN true                                                  |
| }                                                              |
`----------------------------------------------------------------'
```

C.5. Structure

This section provides assistance in implementing a process that verifies whether a candidate string complies with the structure rules applicable to a Frogans address.

The functions provided below do not perform verifications concerning the asterisk character in the Frogans address Section 4.1, as they do not present any particular implementation difficulties.

These functions directly verify whether a candidate string complies with the structure rules applicable to network names. The same functions can be used to verify both network names and site names.

Four functions are required to implement this process:

FUNCTION |c5_verify_structure_network_name|

Description:
This is the main function for this process.

This function checks whether the structure of the candidate string representing a network name is valid.

First it checks whether the candidate string contains the U+002A ASTERISK character. If it does, then the candidate string is rejected.

Otherwise, it checks whether the first character of the candidate string is an unauthorized character. If it is, then the candidate string is rejected.

Otherwise, it checks whether the candidate string contains any connector characters, and if so, whether they follow the rules for connector characters. If the candidate string contains connector characters that do not follow the rules, then the candidate string is rejected.

Otherwise the candidate string is accepted.

Called by:
- none

Calls:
- |c5_verify_first_character|
- |c5_verify_connector_characters|

IFAP lookup tables used:
- none

Input:
- codepoints: a LIST data object containing code points that represent a candidate string containing a network name.

Returns:
true if the structure of the candidate string is accepted, or false otherwise.

Comments:
none

Pseudocode:

```
,-------------------------------------------------------------.
| FUNCTION c5_verify_structure_network_name (codepoints)      |
| {                                                           |
|   VAR index                                                 |
|   VAR cur_cp                                                |
|   FOR index = 0 TO (codepoints.COUNT - 1)                   |
|   {                                                         |
|     cur_cp = codepoints.GET (index)                         |
|     IF (cur_cp == U+002A)                                   |
|     {                                                       |
|       RETURN false                                          |
|     }                                                       |
|   }                                                         |
|   if (CALL c5_verify_first_character (codepoints) == false) |
|   {                                                         |
|     RETURN false                                            |
|   }                                                         |
```

```
|   if (CALL c5_verify_connector_characters (codepoints)      |
|                                                  == false) |
|   {                                                        |
|     RETURN false                                           |
|   }                                                        |
|   RETURN true                                              |
| }                                                          |
`------------------------------------------------------------'
```

FUNCTION |c5_verify_first_character|

Description:
This is a sub-function of the structure verification process.

This function checks whether the first character in the candidate string is a combining characters, a decimal number, or one of five unauthorized characters. If so, the first character in the candidate string is rejected. Otherwise the first character in the candidate string is accepted.

Called by:
- |c5_verify_structure_network_name|

Calls:
none

IFAP lookup tables used:
- LT06_Combining_Marks
- LT10_Decimal_Numbers

Input:
- codepoints: a LIST data object containing code points that represent a candidate string.

Returns:
true if the first character of the candidate string is accepted, or false.

Comments:
none

Pseudocode:

```
,------------------------------------------------------------.
| FUNCTION c5_verify_first_character (codepoints)            |
| {                                                          |
|   TABLE table_LT06                                         |
|   TABLE table_LT10                                         |
```

```
|   VAR first_cp                                                |
|   first_cp = codepoints.GET (0)                               |
|   IF (table_LT06.CONTAINS (first_cp))                         |
|   {                                                           |
|     RETURN false                                              |
|   }                                                           |
|   IF (table_LT10.CONTAINS (first_cp))                         |
|   {                                                           |
|     RETURN false                                              |
|   }                                                           |
|   IF ((first_cp == U+0375) OR                                 |
|       (first_cp == U+05F3) OR                                 |
|       (first_cp == U+05F4) OR                                 |
|       (first_cp == U+06FD) OR                                 |
|       (first_cp == U+06FE))                                   |
|   {                                                           |
|     RETURN false                                              |
|   }                                                           |
|   RETURN true                                                 |
| }                                                             |
`---------------------------------------------------------------'
```

FUNCTION |c5_verify_connector_characters|

Description:

This is a sub-function of the structure verification process.

This function examines each character in the candidate string to see if it is a connector character and if so, whether it complies with the following three conditions: it cannot be the first character nor the last character in the candidate string, it cannot be followed directly by another connector character, and it cannot be followed directly by a combining character.

If each connector character in the candidate string does not meet all three of these conditions, then the candidate string is rejected. Otherwise, the candidate string is accepted.

Called by:

- |c5_verify_structure_network_name|

Calls:

- |c5_is_connector_character|

IFAP lookup tables used:

- LT06_Combining_Marks

Input:
- codepoints: a LIST data object containing code points that represent a candidate string.

Returns:
true if the candidate string is accepted, or false otherwise

Comments:
<1> reject candidate string if connector character is followed by a combining mark

Pseudocode:

```
,-------------------------------------------------------------.
| FUNCTION c5_verify_connector_characters (codepoints)        |
| {                                                           |
|   TABLE table_LT06                                          |
|   VAR index                                                 |
|   VAR cur_cp                                                |
|   VAR previous_cp                                           |
|   cur_cp = codepoints.GET (0)                               |
|   IF (CALL c5_is_connector_character (cur_cp) == true)      |
|   {                                                         |
|     RETURN false                                            |
|   }                                                         |
|   previous_cp = cur_cp                                      |
|   cur_cp = codepoints.GET (codepoints.COUNT - 1)            |
|   IF (CALL c5_is_connector_character (cur_cp) == true)      |
|   {                                                         |
|     RETURN false                                            |
|   }                                                         |
|   FOR index = 1 TO (codepoints.COUNT - 1)                   |
|   {                                                         |
|     cur_cp = codepoints.GET (index)                         |
|     IF (CALL c5_is_connector_character (previous_cp)        |
|                                          == true)           |
|     {                                                       |
|       IF (CALL c5_is_connector_character (cur_cp)           |
|                                          == true)           |
|       {                                                     |
|         RETURN false                                        |
|       }                                                     |
|       IF (table_LT06.CONTAINS (cur_cp))              <1>    |
|       {                                                     |
|         RETURN false                                        |
|       }                                                     |
|     }                                                       |
|     previous_cp = cur_cp                                    |
```

```
|   }                                                          |
|   RETURN true                                                |
| }                                                            |
`--------------------------------------------------------------'
```

FUNCTION |c5_is_connector_character|

Description:
This is a sub-function of the structure verification process.

This function checks whether a code point is a connector character.

Called by:
- |c5_verify_connector_characters|

Calls:
none

IFAP lookup tables used:
none

Input:
- a_codepoint: a code point.

Returns:
true if a_codepoint is a connector character, or false otherwise.

Comments:
none

Pseudocode:

```
,--------------------------------------------------------------.
| FUNCTION c5_is_connector_character (a_codepoint)             |
| {                                                            |
|   IF ((a_codepoint == U+002D) OR                             |
|       (a_codepoint == U+00B7) OR                             |
|       (a_codepoint == U+0F0B) OR                             |
|       (a_codepoint == U+30FB))                               |
|   {                                                          |
|     RETURN true                                              |
|   }                                                          |
|  RETURN false                                                |
| }                                                            |
`--------------------------------------------------------------'
```

C.6. Reference form

This section provides assistance in implementing a process that converts a valid Frogans address (hence in NFKC form) to its reference form, where each character is case folded.

Five functions are required to implement this process:

FUNCTION |c6_generate_reference_form|

Description:
This is the main function for this process.

It generates the reference form of a candidate string by applying the string transformation procedure used in the process of caseless matching for identifiers defined in the Unicode Standard.

First it applies NFD normalization to the input string.

Then it performs NFKC case folding on the code points in the NFD normalized string.

Finally it performs NFC normalization on the case-folded string.

Called by:
none

Calls:
- |c6_normalize_nfd|
- |c6_normalize_nfc|

IFAP lookup tables used:
- LT11_NFKC_Case_Folding

Input:
- codepoints: a LIST data object containing code points that represent the address for which the reference form is required

Returns:
the reference form string

Comments:
none

Pseudocode:

```
,-------------------------------------------------------------.
| FUNCTION c6_generate_reference_form (codepoints)            |
| {                                                           |
|   TABLE table_LT11                                          |
|   LIST work_cps                                             |
|   LIST temporary_cps                                        |
|   LIST nfkc_folded_cps                                      |
|   VAR index                                                 |
|   VAR cur_cp                                                |
|   work_cps = CALL c6_normalize_nfd (codepoints)             |
|   FOR index = 0 TO (work_cps.COUNT - 1)                     |
|   {                                                         |
|     cur_cp = work_cps.GET (index)                           |
|     IF (table_LT11.CONTAINS (cur_cp))                       |
|     {                                                       |
|       nfkc_folded_cps = table_LT11.LOOKUP                   |
|                           (cur_cp, #nfkc_folded_code_point)|
|       temporary_cps.APPEND (nfkc_folded_cps)                |
|     }                                                       |
|     ELSE                                                    |
|     {                                                       |
|       temporary_cps.APPEND (cur_cp)                         |
|     }                                                       |
|   }                                                         |
|   work_cps = CALL c6_normalize_nfc (temporary_cps)          |
|   RETURN work_cps                                           |
| }                                                           |
`-------------------------------------------------------------'
```

FUNCTION |c6_normalize_nfd|

Description:

This is a sub-function of the process for generating the reference form.

The function applies a two-step procedure to generate an NFD normalized string from an input string of code points.

The first step in the two-step procedure is performed by calling the |c6_decompose_canonical function| described below.

The second step in the two-step procedure is performed by calling the |c2_reorder| function described previously.

Called by:
- |c6_generate_reference_form|

Calls:
- |c6_decompose_canonical|
- |c2_reorder|

IFAP lookup tables used:
- none

Input:
- codepoints: a LIST data object containing code points representing the string to be normalized

Returns:
the NFD normalized string

Comments:
none

Pseudocode:

```
,-------------------------------------------------------------.
| FUNCTION c6_normalize_nfd (codepoints)                      |
| {                                                           |
|   LIST work_cps                                             |
|   work_cps = codepoints                                     |
|   work_cps = CALL c6_decompose_canonical (work_cps)         |
|   work_cps = CALL c2_reorder (work_cps)                     |
|   RETURN work_cps                                           |
| }                                                           |
`-------------------------------------------------------------'
```

FUNCTION |c6_normalize_nfc|

Description:
This is a sub-function of the process for generating the reference form.

The function applies a three-step procedure to generate an NFC normalized string from an input string of code points.

The first step in the three-step procedure is performed by calling the |c6_decompose_canonical function| described below.

The second and third steps in the three-step procedure are performed by calling the |c2_reorder| and |c2_decompose| functions described previously.

Called by:
- |c6_generate_reference_form|

Calls:
- |c6_decompose_canonical|
- |c2_reorder|
- |c2_compose|

IFAP lookup tables used:
- none

Input:
- codepoints: a LIST data object containing code points representing the string to be normalized

Returns:
the NFC normalized string

Comments:
none

Pseudocode:

```
,-----------------------------------------------------------------.
| FUNCTION c6_normalize_nfc (codepoints)                          |
| {                                                               |
|   LIST work_cps                                                 |
|   work_cps = codepoints                                         |
|   work_cps = CALL c6_decompose_canonical (work_cps)             |
|   work_cps = CALL c2_reorder (work_cps)                         |
|   work_cps = CALL c2_compose (work_cps)                         |
|   RETURN work_cps                                               |
| }                                                               |
`-----------------------------------------------------------------'
```

FUNCTION |c6_decompose_canonical|

Description:
This is a sub-function of the process for generating the reference form.

It is part of step 1 in both the two-step procedure for generating an NFD normalized string from an input string of code points, and in the three-step procedure for generating an NFC normalized string from an input string of code points.

This function performs a canonical decomposition on each code point in the input string.

Called by:
- |c6_normalize_nfd|
- |c6_normalize_nfc|

Calls:
- |c6_decompose_canonical_cp|

IFAP lookup tables used:
- none

Input:
- codepoints: a LIST data object containing code points that represent the string to be decomposed

Returns:
a string containing the canonical decomposition of each code point in the input string

Comments:
none

Pseudocode:

```
,-------------------------------------------------------------.
| FUNCTION c6_decompose_canonical (codepoints)                |
| {                                                           |
|   LIST work_cps                                             |
|   LIST temporary_cps                                        |
|   VAR cur_cp                                                |
|   VAR index                                                 |
|   FOR index = 0 TO (codepoints.COUNT - 1)                   |
|   {                                                         |
|     cur_cp = codepoints.GET (index)                         |
|     temporary_cps = CALL c6_decompose_canonical_cp (cur_cp) |
|     work_cps.APPEND (temporary_cps)                         |
|   }                                                         |
|   RETURN work_cps                                           |
| }                                                           |
`-------------------------------------------------------------'
```

FUNCTION |c6_decompose_canonical_cp|

Description:
This is a sub-function of the process for generating the reference form.

It is part of step 1 in both the two-step procedure for generating an NFD normalized string from an input string of

code points, and in the three-step procedure for generating an NFC normalized string from an input string of code points.

This function uses a recursive algorithm to decompose a code point. This requires examining the canonical decomposition of the input code point in IFAP lookup table LT02_Canonical_Mapping. If a code point does not exist in the table, then it is included in the normalized string as it is.

The recursive algorithm in this function is based on the rules set forth in the Unicode Standard [Unicode] section 3.7 Decomposition, D68 canonical decomposition.

Called by:
- |c6_decompose_canonical|
- |c6_decompose_compatibility_cp|. The function calls itself recursively.

Calls:
- |c6_decompose_compatibility_cp|. The function calls itself recursively.

IFAP lookup tables used:
- LT02_Canonical_Mapping

Input:
- a_codepoint: the code point to be decomposed

Returns:
a list of code points representing the decomposed form of the input code point

Comments:
<1> if cur_cp exists in the table, the function calls itself

Pseudocode:

```
,-------------------------------------------------------------.
| FUNCTION c6_decompose_canonical_cp (a_codepoint)            |
| {                                                           |
|   TABLE table_LT02                                          |
|   LIST decomposition_cps                                    |
|   LIST work_cps                                             |
|   VAR cur_cp                                                |
|   VAR index                                                 |
|   IF (table_LT02.CONTAINS (a_codepoint))                    |
|   {                                                         |
|     decomposition_cps = table_LT02.LOOKUP (a_codepoint,     |
```

```
|                                          #canonical_mapping)    |
|     FOR index = 0 TO (decomposition_cps.COUNT - 1)              |
|     {                                                           |
|       cur_cp = decomposition_cps.GET (index)                    |
|       work_cps.APPEND (CALL c6_decompose_canonical_cp           |
|                                                 (cur_cp))  <1> |
|     }                                                           |
|     RETURN work_cps                                             |
|   }                                                             |
|   work_cps.APPEND (a_codepoint)                                 |
|   RETURN work_cps                                               |
| }                                                               |
`-----------------------------------------------------------------'
```