Common Lisp 版 CHISE 実装について

守岡 知彦 *

1 はじめに

Common Lisp 版 CHISE 実装である CL-CHISE[5] の現状について概説する。

CHISE (CHaracter Information Service Environment)[2] は著者が中心となって開発している知識処理的アプローチに基づく文字処理環境である。CHISE は文字に関するさまざまな知識を機械可読な知識表現（文字オントロジー）として記述し、その知識に基づいて文字を処理する仕組みを提供している。これはいわば UCS 等の符号化文字集合に対するメタシステムに相当するものといえる。CHISE ではこうした文字知識処理のための枠組だけでなく、その上で記述された大規模な文字知識データセット（CHISE 文字オントロジー）も提供している。

特に、漢字の部品組合せ方の情報（漢字構造情報）のデータは Web サービス「CHISE IDS 漢字検索」[8] でも利用可能であり、UCS の統合漢字をほぼ網羅する検索サービスとして利用されている。バックエンドとなるグラフストレージ Concord[9] に格納された各オブジェクトの情報は EgT[10] を用いて Web ページとして表示することができる。CHISE IDS 漢字検索の検索結果画面には各文字オブジェクトに対応する EgT 頁 (CHISE-Wiki) がリンクされており、これをクリックすることにより文字の詳細情報を閲覧することができる。CHISE は文字に関連する情報に対して素性名を設定することで簡単に取り込むことができ、EgT が提供する素性値の表示法の設定機能を用いることで、さまざまな情報を容易に取り込むことができる。この CHISE の拡張可能性により、EgT 頁 に少ない手間で HNG/HDIC や古典中国語コーパスなどの漢字に関連するさまざまなデータセットの情報を取り込み外部サービスにリンクできた。この結果、CHISE は漢字に関連するデータを繋ぐためのハブ的な Web サービスの一つとなったが、そのシステムの基盤はおよそ 20 年前に開発した XEmacs CHISE[7] であり、CHISE IDS 漢字検索も EgT も Emacs Lisp という Web アプリケーションを実現するのには向いてない言語で書かれており、本格的な Web サービスを実現する上でさまざまな問題を抱えている。

CHISE の文字モデルは基本的に RDF や IPLD [3] などと同様な有向非巡回グラフ (Directed Acyclic Graph; DAG) に基づくものであり、各種グラフストレージの利用を検討してきたが、現在のところ、基本的な文字処理における低レイテンシー性や CHISE が要求する記述上のいくつかの要求を完全に満たす処理系は見出せていない。

* 国文学研究資料館

このため、これまで CHISE 文字オントロジーのデータを RDF のような標準的な枠組に変換し [11][12] その上で利用可能な Web API を作成したり [14] IPFS のような分権的・分散的なストレージに置く [13][15] といった試みを行ってきたが、いずれも CHISE 文字オントロジーの根幹をなす XEmacs CHISE 附属の define-char 形式の文字定義ファイルを置き換えるものとはならず、それらの生成するために XEmacs CHISE 上で Emacs Lisp のプログラムを動かす必要があり、それに時間がかかることもあり、メンテナンス性が悪化してしまい、完全な解決には至っていない。

現状の XEmacs CHISE を用いた文字オントロジー編集作業を継続しつつ、Web サービスにおける XEmacs CHISE 依存性を除去するためには、現行の define-char 形式の文字定義ファイルを直接読み込むことが可能であり、既存の XEmacs CHISE と同等の機能を提供する新実装を開発することが望ましいと言える。[16] では機械語にコンパイル可能な Common Lisp 処理系のひとつである SBCL (Steel Bank Common Lisp) とデータストアとしてインメモリーデータベースのひとつである Valkey (Redis) を用いた CHISE のバックエンドの再実装を試みについて報告している。本稿ではこの実装の改良とその上に XEmacs CHISE 上で稼働していた Emacs Lisp プログラムの移植の試みについて概説する。

2 CHISE における文字表現

CHISE では『Chaon モデル』と呼ぶ方法によって文字を表現するようになっている。これは汎用符号化文字集合に依存することなく自由に文字を表現するために我々が提案しているもので、表現したい文字に関する知識（文字の性質の集合）の機械可読な表現によって文字を表現し操作する方法である。

Chaon モデルでは、文字を説明するための要素（文字の性質や用例など）を『文字素性』(character feature) と呼ぶ。文字素性としては、部首、画数、部品の組合せ方に関する情報（漢字構造情報）、発音、意味、用例、その他文字処理で必要となる各種情報などが考えられる。

現在、CHISE では文字素性を

1. **CCS 素性**: 文字の ID 等への写像
2. **関係素性**: 文字間の関係
3. **構造素性**: （漢字等における）文字の構成要素（部品）の組み合わせかたの情報
4. **分解素性**: 合成済み文字の（１段階の）分解（合成列/結合文字列）を示すための情報
5. **生産物素性**: ある文字の生産物（文字を組み合わせた単語や漢字部品を組み合わせた漢字）を示すための情報
6. **一般素性**: その他（辞書記述的な）文字の性質

の 6 種に分類している。

1. に属する情報は文字符号に関する処理に用いられる。文字符号に関する処理には符号化と復号化の双方の処理が必要であり、文字の 1. に属する文字素性の値を取り出すことは符号化に相当する。文字符号に関する処理は通常高速性が要求されるので、実装上、復号化のために必要となる

逆引用データベースも用意している。[*1]

また、異体字変換（例えば、伝統的字体の漢字を日本の常用漢字や中国の簡体字に変換するような）処理には 2. に属する情報が用いられる。関係素性は逆関係素性と対になり、片方を設定するとその逆関係素性が自動的に設定される。例えば、文字「学」に関係素性 `<-simplified@JP/Jouyou` [*2]を用い、

学 `<-simplified@JP/Jouyou` 學

という関係を設定した場合、文字「學」に

學 `->simplified@JP/Jouyou` 学

という関係が付与される。

3 既存の CHISE システムの構成

表 1 に既存の CHISE 実装のシステム構成を示す。既存の CHISE は最下層のデータストアとして組み込み型 Key Value ストアの一種である Berkeley DB を用い、それを用いてグラフストレージ Concord を構成し、それを XEmacs CHISE のデータベースバックエンドとしている。

Concord[9] は 2 節で述べた CHISE の文字表現モデルを文字以外のデータに適用可能にするために一般化したもので、データの種類を示す『ジャンル』(genre)[*3]と素性の集合によりオブジェクトを表現する。すなわち、文字オブジェクトは文字 (character) ジャンルに属し、素性オブジェクトは素性 (feature) ジャンルに属する。

EgT[10] は Concord のオブジェクトを可視化するための Web アプリケーションであり、XEmacs CHISE 上で動く Emacs Lisp プログラムによって実装されている。

階層	実装
アプリケーション層	EgT, CHISE IDS 漢字検索 等
言語処理系	XEmacs CHISE
文字データ層	libchise
グラフデータ層	Concord
データストア層	Berkeley DB

表 1　現在の CHISE 実装の階層構造

[*1] この特別扱いのために 1. と 6. を区別しているともいえるが、あまり本質的な差異ではないのかも知れない。
[*2] 定義される文字が値に取る文字に対応する日本の常用漢字であることを示す関係素性
[*3] RDF などにおける型に対応するものと考えることができる。

4　設計

4.1　想定する処理系

Common Lisp の処理系として、ANSI Common Lisp に準拠していることや、ネイティブコードへのコンパイラを備えていること、そして普及度等を考慮し、今回、Steel Bank Common Lisp (SBCL) を用いることとした。SBCL は Unicode (UCS) をサポートしており、文字型は U+0000 〜U+10FFFF という Unicode のコード領域全てを保持することができる。

一方、データストアとしては高速性（単純なアクセス時のレイテンシーの少なさ）を重視し、インメモリデータベースの Redis を利用することにした。Redis は Key Value ストアの一種であり、Berkeley DB からの移行も容易であるといえる。なお、Redis は Version 7.2.* までは自由ソフトウェアであったが、2024 年 3 月 20 日に Redis 社は Redis Version 7.4 以降を不自由なライセンスに変更することを表明したため、当面は Redis 7.2.4 を用いることにしているが、将来的には Redis 7.2.4 からフォークした Valkey を用いることにした。

4.2　システムの構成

現在の CHISE 実装のシステム構成（表 1）に合わせ、表 2 に示す構成とする。[*4]

階層	実装
アプリケーション層	（IDS 関連機能等）
文字データ層	CL-CHISE
グラフデータ層	CL-Concord
データストア層	CL-Redis
言語処理系	SBCL

表 2　Common Lisp ベースの CHISE 実装の階層構造

4.3　関数 define-char の挙動

define-char 形式の文字定義ファイルは XEmacs CHISE の組み込み関数 define-char を並べたものである。関数 define-char は連想リストを引数としてとり、そのキー (car) 部は素性名を示し、値 (cdr) 部は素性値を示し、全体としては定義される文字を表現する素性の集合を示す。ここで、この素性の集合を示す連想リストのことを『文字指定子』(character specification; char-spec) と呼ぶ。

[*4] 今の所、IDS 関連機能等のアプリケーション層に属するプログラムも CL-CHISE に収録されている。

XEmacs CHISE において、define-char は Emacs Lisp の関数の一つであり、その値部においては Emacs Lisp の任意のオブジェクトが取れるが、正規化された形式においては Emacs Lisp でも Common Lisp でも読める形式、すなわち、シンボル、数値、リストが用いられている。このため、Common Lisp 上で関数 define-char を実現すれば XEmacs CHISE 附属の define-char 形式の文字定義ファイルをそのまま読むことができるはずである。ただ、この際、その挙動を XEmacs CHISE のものと同様にする必要がある。

XEmacs CHISE において関数 define-char は、引数で指定された文字指定子に CCS 素性が含まれている場合、各 CCS 素性とその値で示される符号位置で復号 (decode-char) できるか試し、最初の復号結果である文字オブジェクトのオブジェクト ID を定義される文字オブジェクトの ID とみなす。但し、文字指定子に CCS 素性 `=ucs` が含まれている場合、その素性値をオブジェクト ID とする。すなわち、この場合、`=ucs` 素性の値を符号位置とする UCS の抽象文字として解釈される。[*5]

また、ISO-IR（ISO/IEC 2022 の国際登記簿）に登録された文字やその他幾つかの文字のために XEmacs CHISE では builtin-character 領域というものを割り当てており、この場合、まだ define-char されていない場合でも decode-char が成功し、その領域に文字オブジェクト ID が割り当てられるようになっていたが、今回の Common Lisp 化においてはこの機能は省くことにした。

文字指定子に含まれたいずれの CCS 素性でも decode が失敗した場合、0xF0000 から順番に文字オブジェクト ID を割り当てる。ただ、現在の CHISE 文字オントロジーでは約 61 万オブジェクトが使われており、UCS の私用領域の範囲内では収まらないと考えられる。Common Lisp における利便性を鑑みれば、CHISE の文字オブジェクトをなるべく Common Lisp における文字として扱えるようにすることが望ましいと考えられるが、そのままでは約 13 万字分しかない Unicode の私用領域には入りきらないので何らかの工夫が必要である。そこで、とりあえず UCS 抽象文字オブジェクトのみを Common Lisp における文字に対応づけ、それ以外は Concord オブジェクト型で表現することにする。

なお、CHISE 文字オントロジーに収録された文字オブジェクトの内、実際に文字として扱う必要性が高いのは抽象文字粒度〜自体粒度の文字オブジェクトのみといえ、抽象字形粒度〜字形粒度や超抽象文字（字種）粒度の文字オブジェクトは Common Lisp における文字として扱う必要性は低いと考えられる。ちなみに、現在、抽象文字オブジェクトは約 5000 個存在し、これはそのまま文字化しても問題なさそうである。一方、字体粒度の文字オブジェクトは約 25 万個存在するため、その全てを文字化することは不可能である。そこで、使用頻度の高そうなものを 10 万個程度選定することが考えられる。

[*5] すなわち、文字オブジェクト ID の 0〜0xEFFFF は UCS の抽象文字のために予約されている。

5 実装

5.1 使用したソフトウェア

Common Lisp 環境の構築には次のものを用いた：

- Common Lisp 環境セットアップユーティリティーの Roswell[4] [*6] 23.10.14.114 を用い、Common Lisp 処理系として SBCL 2.4.6, Quicklisp（2023-10-21 版）を導入したもの。データストアとしては Valkey 7.2.4, Common Lisp 用のバインディングとして CL-REDIS[1] を導入。
- Debian 12 上でパッケージとして導入した SBCL 2.2.9 を用い、Quicklisp（2023-10-21 版）を導入したもの。データストアとしては Redis 7.0.15, Common Lisp 用のバインディングとして CL-REDIS[1] を導入。

5.2 CL-Concord

CL-Concord[6] は従来システムにおける Concord (libconcord) と XEmacs CHISE 中の関連する機能を Common Lisp で再実装したものである。データモデルや基本的な構造は従来のものを踏襲しているが、Common Lisp 化にあたり、API は変更している。C（や Emacs Lisp）において構造体としていたものは CLOS (Common Lisp Object System) におけるオブジェクトとしている。

なお、ASDF (Another System Definition Facility) のシステム名は `cl-concord` であり、パッケージ名は `concord` としている。なお、ここでは `(in-package :concord)` した状態を想定し、シンボルの接頭辞 `chise:` を省略する

5.2.1 data-store

data-store クラスは、従来実装における CONCORD_DS 型に相当するもので、データストア（抽象的な Key Value ストア）を示すためのものである。data-store はそこに収録された genre の集合を持ち、データセットを表現するものでもある。このクラスは

```
(defclass data-store ()
  ((location :initform nil :accessor ds-location)
   (genres :initform (make-hash-table)) ))
```

のように、場所を示す `location` というスロットを持つ。また、ジャンルオブジェクトを一意に保つためのハッシュ表 `genres` も持つ。

data-store クラスのメソッド（総称関数）としては、現在のところ、

[*6] https://cecchine69.rssing.com/chan-67679794/article19.html

ds-get-atom　データストア ds のキー key に対応する文字列を返す

ds-set-atom　データストア ds のキー key に文字列 $value$ を設定する

ds-get-list　データストア ds のキー key に対応するリストを返す

ds-set-list　データストア ds のキー key にリスト $value$ を設定する

ds-rpush　データストア ds のキー key に対応するリストの右側にリスト $value$ を追加する

ds-set-members　データストア ds のキー key に集合 $value$ を設定する

ds-get-members　データストア ds のキー key に対応する集合を返す

ds-adjoin　データストア ds のキー key に対応する集合に要素 $value$ が存在しない場合、key で示される集合に要素 $value$ を追加する

ds-intersection　データストア ds のキー key_1, key_2, ... に対応する集合の共通部分を返す

ds-union　データストア ds のキー key_1, key_2, ... に対応する集合の和集合を返す

ds-store-union　データストア ds のキー $dest - key$ にキー key_1, key_2, ... に対応する集合の和集合を格納する

ds-get　データストア ds のキー key に対応する値を返す

ds-del　データストア ds のキー key の項目を削除する

ds-get-object-spec　データストア ds の $genre - name$ で示されるジャンルにある id で示されるオブジェクトが持つ素性の集合（連想リスト）を返す

ds-some-in-feature　データストア ds において、関数 $func$ を $genre - name$ で示されるジャンル内で素性 $feature - name$ を持つ各オブジェクトに適用する。但し、関数 $func$ の返り値が nil であれば終了する。

などを用意している。

データストアの既定値を示すために大域変数 `*default-ds*` と関数 `default-ds` を設けている：

大域変数 `*default-ds*`　データストアの既定値を示す。初期値は `nil` であり、関数 `default-ds` によって初期化される。

関数 `(default-ds)`

データストアの既定値を返す。

大域変数 `*default-ds*` が `nil` の時、`redis-ds` クラスのオブジェクトを生成し、この変数に設定する。

もし、`redis-ds` クラス以外のデータストアを既定値としたい場合は、大域変数 `*default-ds*` にそのデータストアオブジェクトを設定すれば良い。

5.2.2 location

location はデータストアの場所を示すものである。location をどのように表現するかはデータストアの実装に依存するため、location クラスの定義は

```
(defclass location ()())
```

のように空であり、具体的な実装はこれを継承して作る。

5.2.3 Redis 用実装の定義

Redis 用の実装を与えるために `redis-ds` クラスと `redis-location` クラスを定義している。

data-store クラスは抽象的なデータストアを示すためのもので、具体的なデータストアの実装はこれを継承して作る訳であるが、Redis のための data-store 実装を行う `redis-ds` クラスはその一つである。このクラスのオブジェクトが生成される時、後述する `redis-location` クラスのオブジェクトが生成されて `location` スロットに設定される。

data-store クラスと同様に、location は抽象的な場所を示すためのもので、具体的な場所の表現や実装はこれを継承して作る訳であるが、Redis 用の location 実装を行う `redis-location` クラスはその一つである。

```
(defclass redis-location (location)
  ((connection :accessor redis-location-connection)
   (db-number :initform 0
              :accessor redis-location-db-number)))
```

のように、このクラスは `connection` と `db-number` という 2 つのスロットで Redis を用いたデータストアの場所の情報を表現している。前者は CL-REDIS の `redis-connection` オブジェクトであり、後者は Redis のデータベース番号を示す。

5.2.4 genre

genre クラスは、従来実装における CONCORD_Genre 型に相当するもので、データの種類を示す『ジャンル』(genre) を表現するためのものであり、そのジャンルに属するオブジェクトの集合を表現する。genre クラスは

```
(defclass genre ()
  ((name :accessor genre-name
         :initform 'default :initarg :name)
   (ds :accessor genre-ds :initarg :ds)
   (objects :initform (make-hash-table))))
```

のように、名前を示す `name` とデータストアを示す `ds` というスロットを持つ。また、Concord オブジェクトを一意に保つためのハッシュ表 `objects` も持つ。

5.2.5 object

object クラスは Concord オブジェクトを表現するためのものである。object クラスは

```
(defclass object ()
  ((genre :accessor object-genre :initarg :genre)
   (id :accessor object-id :initarg :id) ))
```

のように、ジャンルを示す genre とオブジェクトの ID を示す id というスロットを持つ。これらのスロットは次のアクセサー関数で参照することができる：

アクセサー (object-genre *obj*) オブジェクト *obj* のジャンルを返す。
アクセサー (object-id *obj*) オブジェクト *obj* の ID を返す。

Concord オブジェクトかどうかを判定するために次の型述語が用意されている：

関数 (object-p *obj*)
obj が Concord オブジェクトの時 non-nil を返す。

Concord オブジェクトは次の関数で定義される：

関数 (define-object *genre spec-alist*)
ジャンル *genre* において素性の集合（連想リスト） *spec-alist* を持つオブジェクトを定義し、定義されたオブジェクトを返す。
genre はジャンル名（シンボル）でも genre オブジェクトでも良い。

例 ID 素性 =foo の値がシンボル test-id-1, name 素性の値が文字列 "test-1" である test ジャンルのオブジェクトを生成し、変数 test-obj に代入する。

```
(setq test-obj
      (concord:define-object :test
        '((=foo .  test-id-1)
          (name .  "test-1"))))
→ #.(concord:object :test 0)
```

Concord オブジェクトの read 可能な表現（例えば、上記の #.(concord:object :test 0)）を実現するために、次の関数を用意している：

関数 (object *genre-name id* (&key *ds*))
ジャンルの名前 *genre-name* とキーワード引数 *ds* で示されるジャンルにおいて、*id* で示されるオブジェクトを返す。もし *id* で示されるオブジェクトが存在しない場合は空のオブジェクトを作って返す。
キーワード引数 *ds* が省略された場合、*default-ds* が用いられる。

関数 concord:define-object は素性の集合（連想リスト）によってオブジェクトを定義する関数であったが、素性の集合よってオブジェクトを探す関数も用意している：

関数 (find-object *genre object-spec*)

ジャンル *genre* において素性の集合（連想リスト） *object-spec* を持つオブジェクトを探索し、最初に見つかったものを返す。

なお、*object-spec* は 1 つ以上の ID 素性を含んでいなければならない。

Concord オブジェクトに関して次のような総称関数が用意されている：

総称関数 (object-get *obj feature* (&optional *default-value* &key *recursive*))

オブジェクト *obj* の素性 *feature* の値を返す。

素性値が存在しない場合、キーワード引数 *recursive* に nil でない値が指定されている場合、オブジェクトの親の素性値を探索し、親にも素性値が存在しない場合にはさらにその親へと先祖の素性値を再帰的に探索する。

(*recursive* が指定されている場合においていずれの先祖にも、そうでなければ、指定されたオブジェクトに) 素性値が存在しない場合、引数 *default-value* の値が指定されていればその値を返し、引数 *default-value* が省略されている場合は nil を返す。

例 変数 test-obj に束縛されたオブジェクトの name 素性の値を返す。

```
(concord:object-get test-obj 'name)
```

→ test-1

総称関数 (object-put *obj feature value*)

オブジェクト *obj* の 素性 *feature* に値 *value* を設定する。

例 ジャンル test において、ID 素性 =foo の値がシンボル test-id-1 であるオブジェクトを返す：

```
(concord:object-put
  test-obj 'note "This is sample.")
```

→ "This is sample."

関数 (decode-object *id-feature id* (&key *genre*))

ジャンル *genre* において、ID 素性 *id-feature* の値が *id* であるオブジェクトを返す。

キーワード引数 *genre* が指定されていない場合、default ジャンルが用いられる。

オブジェクトが見つからない場合は nil を返す。

例 ジャンル test において、ID 素性 =foo の値がシンボル test-id-1 であるオブジェクトを返す：

```
(concord:decode-object '=foo 'test-id-1
                       :genre 'test)
```

→ #.(concord:object :test 0)

総称関数 (object-spec *obj*)

オブジェクト *obj* が持つ素性の集合（連想リスト）を返す。

例 変数 test-obj に束縛されたオブジェクトが持つ素性の集合を返す。

```
    (concord:object-spec test-obj)
→ ((=_id .  0)
   (note .  "This is sample.")
   (name .  "test-1")
   (=foo .  test-id-1))
```

なお、ジャンルと ID が同じであれば同じオブジェクトとなる。

```
(eq test-obj
    (concord:decode-object '=foo 'test-id-1
                           :genre 'test))
→ t
```

5.2.6 素性

従来実装では素性を表現するために CONCORD_Feature 型を設け、feature ジャンルのオブジェクトとして扱っていたが、今回の Common Lisp 版では現在のところ、素性専用のクラスを設けず、基本的に素性名を示すシンボルで表現し、素性属性（素性の素性）を扱いたい場合など素性をオブジェクトとして扱いたい場合、feature ジャンルの Concord オブジェクトとすることにした。

現在のところ、素性名を判定するために次のような関数を用意している：

関数 (metadata-feature-name-p *feature-name*)

素性名（シンボル） *feature-name* がメタデータ素性名の場合、non-nil を返す。

関数 (id-feature-name-p *feature-name*)

素性名（シンボル） *feature-name* が ID 素性名の場合、non-nil を返す。

関数 (structure-feature-name-p *feature-name*)

素性名（シンボル） *feature-name* が構造素性名（ideographic-structure（漢字構造記述）等）の場合、non-nil を返す。

関数 (decomposition-feature-name-p *feature-name*)

素性名（シンボル） *feature-name* が分解素性名（合成済み文字の分解法を示す）の場合、non-nil を返す。

関数 (products-feature-name-p *feature-name*)

素性名（シンボル） *feature-name* が生産物素性名の場合、non-nil を返す。

関数 (relation-feature-name-p *feature-name*)

素性名（シンボル） *feature-name* が関係素性名の場合、non-nil を返す。

関数 (`make-reversed-relation-feature-name` *rel-name*)
素性名（シンボル） *rel-name* が関係素性名の場合、逆関係素性名を返す。

関数 (`expand-feature-name` *feature domain*)
素性名（シンボル） *feature* のドメイン *domain* における素性名を返す。

関数 (`feature` *feature-name* (&key *ds*))
素性名（シンボル） *feature-name* に対応する素性オブジェクトを返す。もし *feature-name* で示される素性オブジェクトが存在しない場合は空の素性オブジェクトを作って返す。
キーワード引数 *ds* が省略された場合、`*default-ds*` が用いられる。

関数 (`find-feature` *feature-name*)
素性名（シンボル） *feature-name* に対応する素性オブジェクトを返す。もし *feature-name* で示される素性オブジェクトが見つからなかった場合は nil を返す。

関数 (`some-in-feature` *func feature-name* (&key *genre ds*))
素性名（シンボル）*feature-name* を持つ各オブジェクトに対して、そのオブジェクトと素性値を引数に取る 2 引数の関数 *func* を呼び、その関数の返り値が non-nil になるまで繰り返す。
キーワード引数 *genre* が省略された場合、`default` ジャンルが用いられる。
キーワード引数 *ds* が省略された場合、`*default-ds*` が用いられる。

5.2.7 集合演算

Redis / Valkey の集合 (set) 型に対する集合演算機能を利用するために、生産物素性をこの set 型で扱うことにし、従来、Lisp 言語側の集合演算機能を利用していたものをなるべく Redis / Valkey の集合演算機能を利用するようにすることで高速化を図ることにした。

以下の総称関数はこの目的で設けられたものである：

総称関数 (`object-adjoin` *obj feature item*)
オブジェクト *obj* の生産物素性 *feature* の値に要素 *item* が含まれていない場合、この要素 *item* を追加する。

総称関数 (`store-union-in-feature` *feature-name dest-obj object* (&rest *objects*))
生産物素性の名前 *feature-name* を持つ $object_1$, $object_2$, ... の和集合をオブジェクト *object* に格納する。

関数 (`union-in-feature` *feature-name* (&rest *objects*))
オブジェクト $object_1$, $object_2$, ... の素性名 *feature-name* で示される素性の値の和集合を

返す。

関数 (intersection-in-feature *feature-name* (&rest *objects*))
オブジェクト $object_1$, $object_2$, ... の素性名 *feature-name* で示される素性の値の共通部分を返す。

生産物素性を扱う場合、concord:object-get や concord:object-put, あるいは、chise:get-char-attribute や chise:put-char-attribute を用いると非常に低速になるので注意が必要である。

5.2.8 補助関数

関数 (sequence-list-p *object*)
object が全ての要素が列であるリストの場合非-nil を返す。

関数 (association-list-p *object*)
object が連想リストの場合非-nil を返す。

5.3 CL-CHISE

CL-CHISE[5] は従来システムにおける libchise と XEmacs CHISE における CHISE 関連機能を Common Lisp で再実装したものである。データモデルや基本的な構造は従来のものを踏襲しているが、Common Lisp 化にあたり、API は変更している。C（や Emacs Lisp）において構造体としていたものは CLOS (Common Lisp Object System) におけるオブジェクトとしている。従来の実装に比べて CL-Concord の層が分厚くなったため、cl-chise の層は相対的に薄くなり、character ジャンル専用のラッパー的なものとなっている。

なお、ASDF のシステム名は cl-chise であり、パッケージ名は chise としている。

以下に CHISE の主要関数について述べる（なお、ここでは (in-package :chise) した状態を想定し、シンボルの接頭辞 chise: を省略する）：

関数 (define-char *char-spec*)
文字指定子（素性の集合を示す連想リスト） *char-spec* を持つ文字オブジェクトを定義し、定義された文字オブジェクトを返す。

関数 (find-char *char-spec*)
文字指定子（素性の集合を示す連想リスト） *char-spec* を持つ文字オブジェクトを探索し、最初に見つかった文字オブジェクトを返す。
なお、*char-spec* は 1 つ以上の CCS 素性を含まなければならない。

関数 (char-spec *character*)

文字 *character* が持つ素性の集合（文字指定子；連想リスト）を返す。
character は Lisp の文字型ないしは character ジャンルの Concord オブジェクトである。

関数 (`char-ccs-spec` *character*)
文字 *character* が持つ ID 素性の集合（文字指定子；連想リスト）を返す。
character は Lisp の文字型ないしは character ジャンルの Concord オブジェクトである。

関数 (`decode-char` *ccs code-point*)
CCS 素性 *ccs* の符号位置が *code-point* である文字を返す。
返される文字が UCS の抽象文字である場合、Lisp の文字型になる。そうでない場合、character ジャンルの Concord オブジェクトで返される。

関数 (`encode-char` *character ccs*)
文字 *character* の CCS 素性 *ccs* の値（符号位置）を返す。
character は Lisp の文字型ないしは character ジャンルの Concord オブジェクトである。

関数 (`get-char-attribute` *character attribute* (&optional *default-value*))
文字 *character* の素性 *attribute* の値を返す。
素性値が存在しない場合、引数 *default-value* の値が指定されていればその値を返し、引数 *default-value* が省略されている場合は `nil` を返す。
character は Lisp の文字型ないしは character ジャンルの Concord オブジェクトである。
この関数は `char-feature` と異なり素性値の再帰的探索を行わない。

関数 (`char-feature` *character feature* (&optional *default-value*))
文字 *character* の素性 *feature* の値を返す。
素性値が存在しない場合、引数 *default-value* の値が指定されていればその値を返し、引数 *default-value* が省略されている場合は `nil` を返す。
character は Lisp の文字型ないしは character ジャンルの Concord オブジェクトである。
この関数は `get-char-attribute` と異なり素性値を先祖側に再帰的に探索する。

関数 (`put-char-attribute` *character attribute value*)
文字 *character* の素性 *attribute* の値を *value* に設定する。
character は Lisp の文字型ないしは character ジャンルの Concord オブジェクトである。

関数 (`normalize-as-char` *object*)
文字オブジェクト *object* が UCS の抽象文字であり、Common Lisp における文字型オブジェクトで表現可能な場合に、文字型オブジェクトに変換する。そうでない場合、Concord の文字クラスのオブジェクトを返す。

5.3.1 生産物素性用関数

下記の関数は生産物素性に対する集合演算を Redis / Valkey 側で実行するために用意された CL-Concord の（総称）関数を character ジャンル専用にしたラッパー関数である。

関数 (`adjoin-char-attribute` *character attribute item*)
文字 *character* の生産物素性 *feature* の値に要素 *item* が含まれていない場合、この要素 *item* を追加する。
character は Lisp の文字型ないしは character ジャンルの Concord オブジェクトである。
Note この関数は `concord:object-adjoin` の文字専用版である。

5.3.2 JSON

Web アプリケーションを作るときにおける JavaScript プログラムとのやりとりなどでは JSON 形式が便利である。このため、Concord の object-spec / CHISE の char-spec 形式のデータを JSON で出力するための機能を提供している。

関数 (`encode-json` *obj* (&optional *stream*))
Lisp オブジェクト *obj* を JSON に変換して出力する。
省略可能な引数 *stream* は出力するストリームを示し、その規定値は変数 `json:*json-output*` の値（規定値は標準出力）である。

これは `cl-json` を利用したもので、これに Concord オブジェクト用のメソッドを追加するとともに、CHISE に合わせてシンボルの変換法を修正している。

6 既存のアプリケーションの移植

「CHISE IDS 漢字検索」や EgT 等の CHISE 関連 Web サービスや XEmacs CHISE 上で利用するアプリケーション等は Emacs Lisp で書かれている。Emacs Lisp と Common Lisp はともに初期 Lisp 主要方言のうちの MacLisp に属しており、その構文は比較的似通っているものの、Emacs Lisp がテキストエディターのためのドメイン固有言語であることから、ファイルをバッファーと呼ばれるデータ構造に一括して読み込んで操作するというモデルを採っているのに対して、Common Lisp ではストリームという抽象化が行われているという根本的な違いがある。また、Common Lisp ではループなどの制御構造のための多様な構文が用意されているのに対し、Emacs Lisp では特殊形式 `while` を用いる。[*7]文字やベクトル等の（定数の）書き方も Common Lisp と Emacs Lisp では異なり、また、整形 (format) のための関数の仕様も全く異なっている。

[*7] 但し、Emacs Lisp には Common Lisp の機能をエミュレートするためのパッケージが存在し、Common Lisp 風に記述することも可能である。

また、変数の束縛に関するセマンティクスとして、Common Lisp では局所変数は静的束縛になっているのに対し Emacs Lisp では動的束縛になっているという点もある。但し、大域変数はどちらも動的束縛である。Emacs Lisp においても局所変数が動的束縛であることを利用したプログラムを書くのは行儀が良いとはいえず、大域変数の挙動を局所的にいじるような使い方が多いと思われるが、一定の注意は必要かもしれない。

こうしたことを鑑みれば、文字列処理に関するプログラムは比較的少ない手間で移植できると思われるのに対し、バッファーを利用したプログラムはほぼ新規実装になると思われる。

6.1 Emacs Lisp 互換機能の実装

6.1.1 制御構造

ループ構文は数が多くまた Common Lisp のループ構文は複雑で種類も多いため人手での書き換えが面倒である。このため、Emacs Lisp のループ用特殊形式 `while` を Common Lisp のマクロとして実装し、CL-Concord に追加した。なお、`while` は CL-CHISE においても CL-Concord から import しているため、`chise` パッケージ内でも `concord:` 接頭辞なしに利用できる。

マクロ （`while` *test* &body *body*）

式 *test* の評価値が nil でない間、*body* を繰り返す。

6.1.2 文字定数

Emacs Lisp の文字定数は `?a` のように「`?`」の後に文字を書く形式であるのに対し、Common Lisp では `#\a` のように「`#\`」の後に文字を書く形式になっているという違いがある。また、エスケープ記法も異なり、Common Lisp には Emacs Lisp にない文字の名前を利用したエスケープ記法なども存在する。

一方、XEmacs CHISE のファイル入出力の際の文字列変換では `&`*foo*`;` という SGML 風の実体参照形式の復号機能を提供しており、編集中の文字定義ファイルでは Emacs Lisp の文字定数形式と実体参照形式を組み合わせたものが出現することがある。後述するように、SGML 風の実体参照形式は IDS ファイルにも出現するため、この復号処理を行う関数を実装するとともに、文字定義ファイルの読み込みを行うインストラーに Emacs Lisp の文字定数をサポートするためのリードマクロを設定した。

関数 （`read-entity-reference` *in*）

入力ストリーム *in* の先頭に「&」があれば「;」まで読み込み実体参照形式として解析を行う。但し、「;」を見つける前に、空白やタブ、改行文字、復帰文字を読み込むかストリームの終端に到達した場合、それまでに読み込んだ文字列をそのまま返す。「&」から「;」までの実体参照形式の候補がうまく復号できない場合も同様である。

実体参照形式として復号が成功した場合、その結果が UCS の抽象文字であり Common

Lisp の文字型で表現できる場合は文字型として返す。そうでない場合は、character ジャンルの Concord オブジェクトとして返す。

Emacs Lisp の文字定数を読み込むためのリードマクロは以下の通りである：

```
(set-macro-character
 #\?
 #'(lambda (stream char)
     (or (read-entity-reference stream)
         (read-char stream nil))))
```

6.2 CHISE-IDS の移植

XEmacs CHISE の文字オントロジーを構成するためには、XEmacs CHISE 附属の文字定義ファイルを読み込んだ状態で、CHISE 漢字構造情報データベース (CHISE-IDS) の IDS 形式の漢字構造記述ファイルを読み込む必要がある。また、CHISE-IDS ではインストール時に、IDS 形式の漢字構造記述ファイルを読み込み、機能的漢字構造を表現する `ideographic-structure` 素性と皮相漢字構造を表現する `ideographic-structure@apparent` 素性に情報を追加するだけでなく、機能的漢字構造を皮相漢字構造に変換したり、「⿰女公羽」を「⿱女翁」に変換するといった中間部品の簡約処理を行っているが、IDS ファイルを読み込んで IDS 形式のデータを構文解析し素性値として設定したり、漢字構造を変換するといった処理は Emacs Lisp のプログラムで実現されており、CL-CHISE で同様のことを実現するためには既存のプログラムを移植するか等価な処理を行う新規実装を実現する必要がある。

こうした処理のうちややこしい部分は概ね文字列やリストの処理で実現されており、比較的移植が簡単である。一方、ファイルを読み込む部分は新規実装が必要となるが比較的単純であり手間は少ない。

6.3 ids.lisp

`ids.lisp` は CHISE-IDS 附属の Emacs Lisp プログラムの内、

- ids.el が提供する IDS の構文解析処理
- ids-read.el が提供する IDS ファイルの読み込み機能
- ids-find.el が提供する検索用インデックスの作成処理、漢字構造変換処理、検索機能（Web UI は除く）

を Common Lisp で実装したものである。その大半は既存の Emacs Lisp プログラムを移植したものである。

ids.el の構文解析器は文字を入力としていたが、この Common Lisp 化に当たってはストリームを入力とするようにした。Common Lisp のストリームは文字列でもファイルでもネットワークで

もそのまま利用でき、文字列以外の入力を文字列に変換する必要がなくなり効率的である。但し、ストリームから取り出した文字は 1 文字しかストリームに戻せないが、解析のバックトラックを実現するためには複数文字を戻す必要があるため、前の処理で戻した文字列と入力ストリームの組を（ids.el における構文解析木の入力文字列の代わりとなる）入力とすることにした。また、XEmacs CHISE のファイル入出力の際の文字列変換では `&`*foo*`;` という SGML 風の実体参照形式の復号機能を提供しており、CHISE-IDS の IDS ファイルでも使用されているため、入力ストリーム中の実体参照を構文解析して復号結果を返す関数 `read-entity-reference` を実装した。

関数 （`ids-parse-string` *ids-string*）
文字列 *ids-string* を IDS として構文解析し、その結果となる spec-alist を返す。

関数 （`ids-parse-element` *previous in*）
入力ストリーム *in* を IDS として構文解析し、その結果を主値に返し、入力ストリーム *in* から読み込んだものの余ってしまった文字列を副値に返す。
引数 *previous* は先にこの関数を実行して余ってしまった文字列を渡すためのものであり、最初に実行する際は `nil` を指定する。

関数 （`ids-read-file` *file* (&key *override prompt*)）
IDS ファイル *file* を読み込み、CHISE 文字オントロジーに登録する。
キーワード引数 *override* に nil でない値が指定されている場合、対象文字の `ideographic-structure` 素性や `ideographic-structure@apparent` 素性に素性値が存在していても IDS ファイルの該当項目が IDS として解析に成功した場合、その解析結果で上書きする。そうでない場合、既存の素性値が優先され IDS ファイルの該当項目は無視される。
キーワード引数 *prompt* に nil でない値が指定されている場合、標準出力にメッセージが出力される。

関数 （`ids-update-index` (&optional *s*)）
漢字構造記述データ（`ideographic-structure` 素性、および、それにドメイン `@apparent`, `@apparent/leftmost`, `apparent/rightmost` を付けたもの）を元に検索用インデックス（`ideographic-products` 素性）を作成する。
省略可能な引数 *s* はメッセージを出力するためのストリームである。省略された場合、標準出力が用いられる。

関数 （`ideograph-find-products` *components*）
引数 *components* で指定された部品を全て持つ漢字のリストを返す。但し､引数 *components* は文字列か文字（Common Lisp の文字型データか character ジャンルの Concord オブジェクト）のリストでなければならない。

例　`(chise:ideograph-find-products "犹日")`

→ (#\U27B82 #\U2C582 #\U2110B #\U30F00 #\U6FF3 #\U3B31 #\U233AF #\U28BE9
#\U305BD #\U25333 #\U39A7)

関数 (ideographic-structure-some-chars *func structure* (&key *require-component*))
漢字構造のパターン *structure* にマッチする全ての漢字に対して関数 *func* を適用する。但し、関数 *func* が nil を返すとその時点で終了する。
structure は (#\□ #\車 #\多) のような IDS をリストにしたものでも、(#\□ #\車 #\x) のように変数を含んだものでも良い（ASCII 文字は変数として扱われる)。
キーワード引数 *require-component* が指定されている場合、*structure* は少なくとも一つは変数でない部品を含まなければならない。そうでない場合、(#\□ #\x #\x) のような変数だけの部品も認められる（但し、低速である)。

関数 (ideographic-structure-find-chars *structure*)
漢字構造のパターン *structure* にマッチする全ての漢字のリストを返す。
structure は (#\□#\車 #\多) のような IDS をリストにしたものでも、(#\□#\車 #\x) のように変数を含んだものでも良い（ASCII 文字は変数として扱われる)。但し、少なくとも一つは変数でない部品を含まなければならない。

例
```
(chise:ideographic-structure-find-chars
  (cdr (assoc 'chise:ideographic-structure
  (chise:ids-parse-string "□車□ xx"))))
```
→ (#\U282BF #\U283AA #\U4854 #\U8F1A #\U8F1F #\U28339 #\U2C9FA)

関数 (ids-find-chars-including-ids *structure*)
漢字構造 *structure* を部品として持つ漢字のリストを返す。
structure は文字（Common Lisp の文字型、もしくは、character ジャンルの Concord オブジェクト）か (#\□ #\車 #\多) のような IDS をリスト化したものである。

関数 (ideographic-structure-compact *structure*)
漢字構造 *structure* を簡約化する。
structure は (#\□ #\車 #\多) のような IDS をリスト化したものである。

関数 (functional-ideographic-structure-to-apparent-structure *structure*)
機能的漢字構造 *structure* を皮相漢字構造に変換した結果を返す。
structure は (#\□ #\放 #\小) のような IDS をリスト化したものである。

例
```
(format t "~a" (functional-ideographic-structure-to-apparent-structure
                '(#\□ #\放 #\小)))
```
→ (#\□ #\方 #\尒)

7 インストール

7.1 準備 (1) Redis / Valkey

CL-Concord および CL-CHISE を利用するためにはあらかじめ Redis か Valkey を導入し、いずれかのサーバーを動かしておく必要がある。

Mac の場合 （Homebrew を使っている場合）

% brew install valkey

% brew services start valkey

Debian 系 Linux の場合 （Debian や Ubuntu 等）

% apt install redis

7.2 準備 (2) SBCL と Quicklisp

SBCL と Quicklisp をインストールする。

Roswell を用いる場合、

% ros setup

とすれば良い。

Roswell を用いない場合、パッケージを利用するかソースから SBCL を導入し、`https://www.quicklisp.org/beta/` の記述を参考に Quicklisp を導入する。

7.3 ローカルプロジェクト用のパスの調査

sbcl を立ち上げ、変数 `ql:*local-project-directories*` で指されるローカルプロジェクト用のパスを調べる。

(例)
```
% sbcl
* ql:*local-project-directories*
```

Roswell 環境だとデフォルトでは

```
~/.roswell/lisp/quicklisp/local-projects/
```

Roswell を使わない素の SBCL + Quicklisp の場合、デフォルトでは

```
~/quicklisp/local-projects/
```

となっていると考えられる。

7.4 CL-Concord と CL-CHISE のダウンロード

ql:*local-project-directories* で指されるパスに CL-Concord と CL-CHISE をダウンロードする。

(例)
```
% cd ~/quicklisp/local-projects/
% git clone https://gitlab.chise.org/CHISE/cl-concord
% git clone https://gitlab.chise.org/CHISE/cl-chise
```

7.5 CL-Concord と CL-CHISE を Quicklisp に登録

SBCL を立ち上げ、関数 ql:register-local-projects を実行する。

(例)
```
% sbcl
* (ql:register-local-projects)
→ NIL
```

7.6 ql:quickload

SBCL を立ち上げ CL-CHISE を ql:quickload する。

(例)
```
% sbcl
* (ql:quickload :cl-chise)
```

なお、最初に CL-CHISE を読み込んだ時は附属の文字定義ファイルを読み込んだ後、CHISE 漢字構造情報データベースをダウンロードして読み込み、漢字構造変換処理を行うなどして CHISE 文字オントロジーの構築を行うため、ロードに非常に時間がかかることに注意。2 回目以降は Redis もしくは Valkey に構築済みの文字オントロジーを利用するためすぐに起動する。

また、CL-Concord だけを使いたい場合は

(例)
```
% sbcl
* (ql:quickload :cl-concord)
```

とすれば良い。

8 おわりに

Common Lisp 版 Concord 実装 CL-Concord と Common Lisp 版 CHISE 実装 CL-CHISE について述べた。CL-Concord と CL-CHISE は XEmacs CHISE 版と同様の機能を実装すること

ができた。SBCL は Common Lisp のプログラムをネイティブコードにコンパイルして実行することができることと、バックエンドのデータストアにインメモリー型の Valkey / Redis を用いたこともあり、速度的にも概ね現行の実装と遜色のないものになったと考えられる。

現在のところ、XEmacs CHISE 附属の文字定義ファイルと CHISE 漢字構造情報データベースの読み込みと漢字構造変換の処理は実装できているが、今後は HNG/HDIC 等のファイル入力を伴う外部データの取り込みや、Web アプリケーション等の移植・実装を進めたいと考えている。

参考文献

[1] Vsevolod Dyomkin. CL-REDIS — a fast and robust Common Lisp client for Redis. `https://github.com/vseloved/cl-redis`.

[2] Tomohiko Morioka. Multiple-policy character annotation based on CHISE. *Journal of the Japanese Association for Digital Humanities*, Vol. 1, No. 1, pp. 86–106, 2015 年 11 月.

[3] Protocol Labs. IPLD. `https://ipld.io/`.

[4] Roswell - Common Lisp environment setup utility. `https://github.com/roswell/roswell`.

[5] MORIOKA Tomohiko. CL-CHISE. `https://gitlab.chise.org/CHISE/cl-chise`.

[6] MORIOKA Tomohiko. CL-Concord. `https://gitlab.chise.org/CHISE/cl-concord`.

[7] MORIOKA Tomohiko, et al. XEmacs CHISE. `http://www.chise.org/xemacs/`.

[8] 守岡知彦. CHISE IDS 漢字検索. `https://www.chise.org/ids-find`.

[9] 守岡知彦. Concord: プロトタイプ方式のオブジェクト指向データベースの試み. Linux Conference 抄録集, Vol. 4, , 2006 年.

[10] 守岡知彦. Wiki 的手法に基づく構造化データの編集について. 人文科学とコンピュータシンポジウム論文集 —人文工学の可能性 〜異分野融合による「実質化」の方法〜, 情報処理学会シンポジウムシリーズ, 第 2010 巻, pp. 33–40. 情報処理学会, 情報処理学会, 2010 年 12 月.

[11] 守岡知彦. CHISE の階層的素性名の RDF 化の試みについて. 情処研報, Vol. 2013-CH-97, No. 3, pp. 1–6, 2013 年 1 月.

[12] 守岡知彦. CHISE の RDF 化の試み. 情処研報, Vol. 2017-CH-114, No. 1, pp. 1–6, 2017 年 5 月.

[13] 守岡知彦. 内容アドレッシングを用いた多粒度漢字構造情報表現の試み. 情報処理学会論文誌, Vol. 61, No. 2, pp. 171–178, 2020 年 2 月.

[14] 守岡知彦. CHISE の Web API 化の試み、ついでに、RDF 化四度目の正直? 東洋学へのコンピューター利用 第 33 回研究セミナー, pp. 69–87, 2021 年 3 月.

[15] 守岡知彦. 漢字構造検索機能の IPFS 化の試み. じんもんこん 2023 論文集, pp. 161–168. 情報処理学会, 情報処理学会, 2023 年 12 月.

[16] 守岡知彦. Common Lisp を用いた CHISE の再実装の試み. 情処研報, Vol. 2024-05-11, No. 14, pp. 1–7, 2024 年 5 月.