記 号 処 理 61－4
（1991. 9. 13）

Lispによる漢字フォント作成支援ツールの開発

石井 裕一郎
東京大学 工学部

我々は漢字フォントを骨組みデータから自動的に生成するという研究を進めている。漢字を構成する基本的な骨組みデータは人間が作りあげなければならない。自動生成のアルゴリズムのチェックも人間が行なう。そこで作業が円滑にすすめられるような環境が必要になる。ここではLispを用いて作成した漢字フォント作成支援ツールの開発について述べる。漢字フォント作成支援ツールはUtiLispとULXで書かれており、漢字フォントを高速で効率よく開発するのに役立っている。

Tools for Creating Kanji Skeleton Fonts with Lisp

ISII Yuitirou
Faculty of Engineering, University of Tokyo
7-3-1, Hongo, Bunkyo-ku, Tokyo, 113 Japan

We are developing algorithms which automatically generate Kanji (Japanese characters) fonts in many styles. Each Kanji is described by the combination of some primitive components. The programmers must specify skeleton data of such components, and examine the algorithms. It is necessary to provide some tools which helps the programmers work smoothly and we introduce them in this paper. They are implemented with UtiLisp and ULX(UtiLisp X Interface) library, and work fast enough to invent Kanji fonts efficiently.

1 はじめに

我々は日本語の「美しい」文字フォントを段階を経て自動的に生成する漢字スケルトンフォントプロジェクトを進めている [1]. 漢字スケルトンフォントでは, 複雑な漢字は自動的に生成されるが, もっとも基本となる骨組みのデータは人間が作り上げなければならない. また, 自動的に生成するアルゴリズムの妥当性の検査も人間がおこなう. この際に, 人間が作業を進めやすいようにする環境が必要となる. 漢字フォント作成支援ツールはこの環境を構成する部品の一部でスケルトンエディタと組合せエディタからなっている. このようなエディタには他に [2] などがある.

漢字スケルトンフォントプロジェクトでは開発言語としてUtiLisp[3, 4] を用いている. 漢字フォント作成支援ツールもまたUtiLispで記述されており, X-Window 上で動く. X をUtiLispから利用するためにULX(UtiLisp X Interface) を使っているが, これは CLX (Common Lisp X Interface)[5] を移植したものである.

漢字フォント作成支援ツールは実際に骨組みデータを編集する作業に使われており, フォント作成を効率よく進めるのに役だっている.

ここでは, この漢字フォント作成支援ツールの特徴と開発時に留意した点について述べる.

2 漢字スケルトンフォントの構成

漢字フォントは次の 3 段階から作られる.

エレメント 右払い, 縦棒, 左払い, 横棒などスケルトンフォントを構成する最小単位で 16 種類ある. それぞれ決まった数の制御点を持つ.

プリミティブ エレメントの集合にエレメントの間の連結情報を加えたものである. 部首に相当する. 単独で漢字フォントになるものもある.

複合パーツ プリミティブや複合パーツを組み合わせたものである. 複合パーツが漢字フォントになる. プリミティブや複合パーツを拡大縮小や移動して組み合わせて複合パーツを作る. これは組み合わせアルゴリズムにしたがう.

このうち, プリミティブを作成 / 編集する作業と組み合わせアルゴリズムで使われるプリミティブの属性を決める作業は人間がおこなう. そこで, この作業を支援するツールが必要となる. このツールがそれぞれ, スケルトンエディタと組み合わせエディタである.

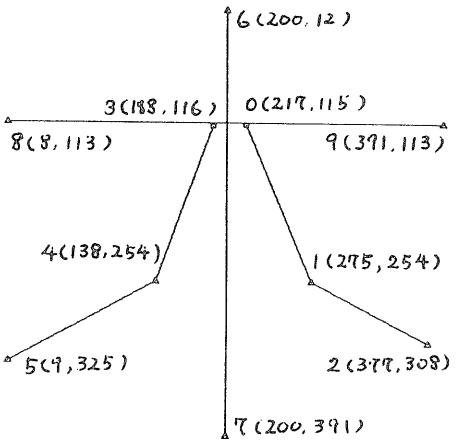

図 1: プリミティブ「木」

プリミティブは次の例のような構成になっている (図 1).

```
(setq 木 '(((217 115 (link-ok t))
            (275 254)
            (377 308)
            (188 116 (link-ok t))
            (138 254)
            (9 325)
            (200 12)
            (200 391)
            (8 113)
            (391 113))
           ((migi (0 1 2))
            (hidari (3 4 5))
            (tate (6 7))
            (yoko (8 9)))
           (xunit . 150)
           (yunit . 150)))
```

データの car が制御点の位置を, cadr がエレメントを, cddr が組み合わせ属性を表している.

漢字フォントは次の例のような構成になっている.

```
(setq 腕 '(yoko2 にくづき 宛))
(setq 宛
   '(tate2 うかんむり
           (yoko2 夕
                  ふしづくり)))
```

データの car が組み合わせの種類を, cdr が組み合わせるプリミティブを表している. yoko2

は横方向の組合せ, tate2 は縦方向の組合せをしめす.

3 UtiLisp X Interface

フォント作成の作業は視覚的なものである. 編集対象を目で見て形がよいかどうか判断しながら作業を進めていく. したがって, 絵を表示する機能が開発言語に必要となるのだが, UtiLispではULXというライブラリを利用することができる. ULXは CLX(Common Lisp X Interface) をUtiLispに移植したものである.

3.1 ULXの特徴

CLX は X で利用できるサービスをすべて網羅しており, ソケット通信のための関数以外はすべて Lisp で記述されている. ULXでもこれは変わらない.

ULXの特徴は, 高速なことである. これはUtiLispが軽く高速な処理系であることから予想される結果であるが, 十分速いためにエディタのプログラムをコンパイルする必要がまったくない. これがエディタを開発する効率をあげるのに役立っている. またULXを利用すると, Common Lisp の特殊形式, マクロ, 構造体 (if, when, let*, dolist, defstruct など) がUtiLispでも使えるようになる.

3.2 ULXでのイベントの取扱い

マウスのボタンを押したり放したり, マウス (ポインタ) を動かしたりキーボードを打ったり, ウィンドウが隠れたり現れたり, ポインタがウィンドウを出入りしたときに, X ではイベントが発生する. イベントによって人間と通信をおこなうのである. イベントで得ることのできる情報は,

- イベントが発生したウィンドウ
- イベントが発生したときのポインタの位置
- 押した / 放したボタンの種類

などである. イベントが発生したウィンドウはULXでは構造体で表現している.

3.3 ULXの現状と今後

ULXは大変に高速なライブラリで, 対話的に利用しているとC言語でXを利用するプログラムを書くよりもはるかに楽に開発が進められる.

ポインタを動かすとそれに追随して図形も動く, というようなことも自然なスピードでおこなえる. これは, 視覚的なアプリケーションを作成する際にはぜひとも必要な特質である.

最大の難点は, ウィンドウを作るのが遅い, ということである. 特にメニューをウィンドウで作ると多くのウィンドウが必要になるため, 作業を始める前の処理にたいへん時間がかかる. この欠点を除くための努力が現在も続いている. しかしウィンドウが少ない場合はそれほど問題にはならない.

ULXへの移植はまだ終わっていないため, CLX では対応している X のサービスで利用できないものがある. たとえばULXはモノクロのみであるし, リソースやビットマップにも対応していない. しかし, これらのサービスは必ずしもエディタの開発に必要なものではない. 必要不可欠な部分はすでに完成しているため, それ以上の移植は必要に応じておこなうことになっている.

4 エディタの構成

スケルトンエディタと組合せエディタはいずれも基本的には作業をマウスだけで進められるようになっている. エディタのように人間に近いツールは使い勝手が命である. 漢字スケルトンフォントプロジェクトのエディタは簡単ではあるがわかりやすいユーザインターフェースを提供している. エディタのプログラムのソースは現在合わせて約 6000 行である. 図 2 から図 6 はスケルトンエディタの編集画面である.

4.1 メニュー

メニューはポインタが入ってくると枠線が描かれる. ポインタが出ていくとその枠線が消える. メニューの上にポインタを置いてボタンを押すとそのメニューが選ばれたことになる. 選ばれたメニューは反転する. 実数を入力する際にはスライド抵抗のようなウィンドウを使うことができる. 数直線上にポインタを置いてボタンを押すとそこに指針が動き, 変数の値が変更される.

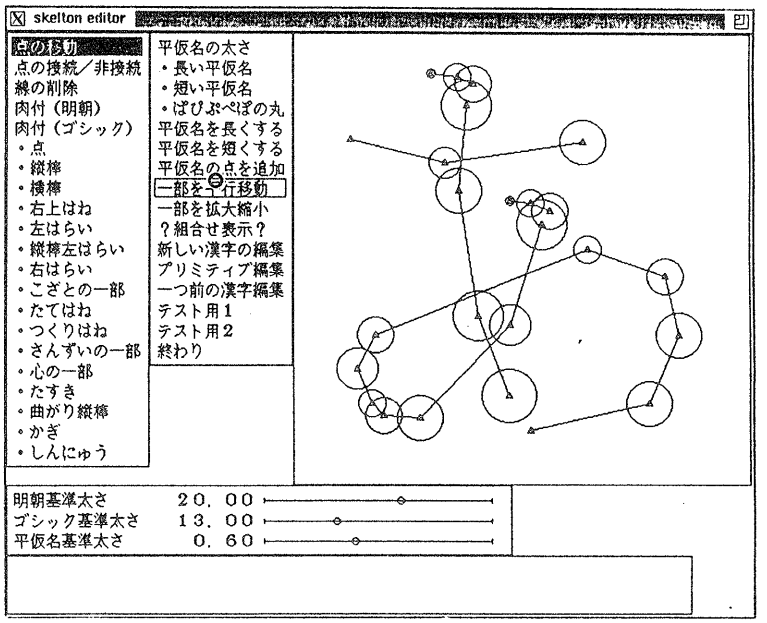

図 2: 編集画面「あ」

4.2 エディットウィンドウ

メニューで選ばれた作業に応じてポインタとボタンを利用してプリミティブや組み合わせ属性を編集できる. たとえば, 制御点の移動は動かしたい制御点の近くへポインタを持っていき, そこでボタンを押して, 押したまま目的のところまでポインタを移動し, ボタンを放したところに制御点が移る. 制御点を移動している間は, その制御点につながっている線分も一緒に動いていく.

5 UtiLispによる開発

UtiLispを開発言語とした理由は以下の通りである.

- 対話型言語なので, プログラムの変更や修正, 確認が容易である. 図を表示するようなアプリケーションでは実際に表示させてみてアルゴリズムが正しいかどうか検査するという手法をよく用いるが, C言語などを使う場合にはコンパイルに時間がかかり, このような開発がしにくい. 一方対話型言語はこのような手法にたいへん向いている.
- データ構造の変更が容易である. データはすべてS式で表されているため, 開発途中でデータ構造を変更したくなっても簡単に変換することができる. 連想リストを使うことによって多くの情報を無理なく埋め込むこともできる.
- UtiLispの中身がわかっている. 処理系本体に手をいれることのできる者が研究室にいるため, UtiLispの機能上の問題やバグが発生してもすぐに対処できる.

Lisp初心者であった筆者が触りはじめてから1年もたたないのにこのようなプログラムを作成できたのも以上の利点によるものである.

5.1 開発の経緯

当初, UtiLispとXとのインターフェースはC言語を使って関数を定義していたが, これには,

- ライブラリを動的にリンクする必要がある.
- Xlibの各関数に対応するLisp関数を用意しなければならないが, その数がたいへんに多い.
- Xlibで使われているC言語の構造体をUtiLispとやりとりする際の相性が悪い.

という問題があり,

- 400×400ウィンドウが1つだけ作られる.

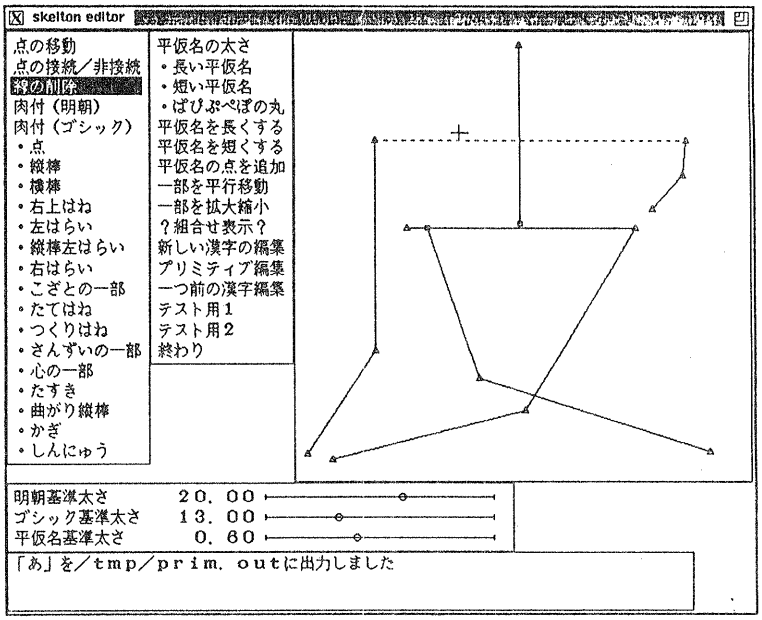

図 3: 編集画面「皮」(線の削除)

- つかまえられるイベントはボタンの押し/放しとキーボードの押し/放しだけである.

のような限定された機能のものとなっていた. このため, 使い勝手も悪くなってしまい,

- エレメントの追加にはキーボードで暗号のようなコマンドを打ち込んでからマウスで指定した. 例えば, つくりではts, ひだりはらいではhなどのコマンドである.
- できる作業が少なく, エレメントの追加と制御点の移動しかできなかった.
- ほかのウィンドウの後ろに隠れると画面から表示が消えてしまった.
- ポインタの動きに追随して描画することができなかった.

のような欠点があった.
ULXを利用することによって

- メニューで作業が選べるようになり, 操作性が向上した. enter-notifyとleave-notifyイベントに対応したため, メニューにポインタがはいると枠線を描くようなことができるようになった.
- 多くの作業に対応した. エレメントの削除やプリミティブの拡大, 縮小など必要に応じて次々に作業メニューを増やせるようになった.
- exposureイベントに対応したため, ウィンドウが隠れて表示が消えても再表示ができるようになった.
- motion-notifyイベントに対応したため, ポインタの動きに追随する描画ができるようになり視覚的訴求力が増した.
- ほかのウィンドウと通信できるようになった. 漢字を入力するときにはkinputを利用することができる.

のように欠点を克服することができた.

5.2 イベント起動

イベントがどこで発生したかはULXのwindow構造体で返される. C++のようなオブジェクト指向言語の発想ではこのwindow構造体から新しい構造体を導出して, イベントの種類に対応した仮想関数や新しいデータをその構造体の定義に加えておくといった手法が使われる. UtiLispではこのような手法は言語仕様には含まれていない. そこで, window構

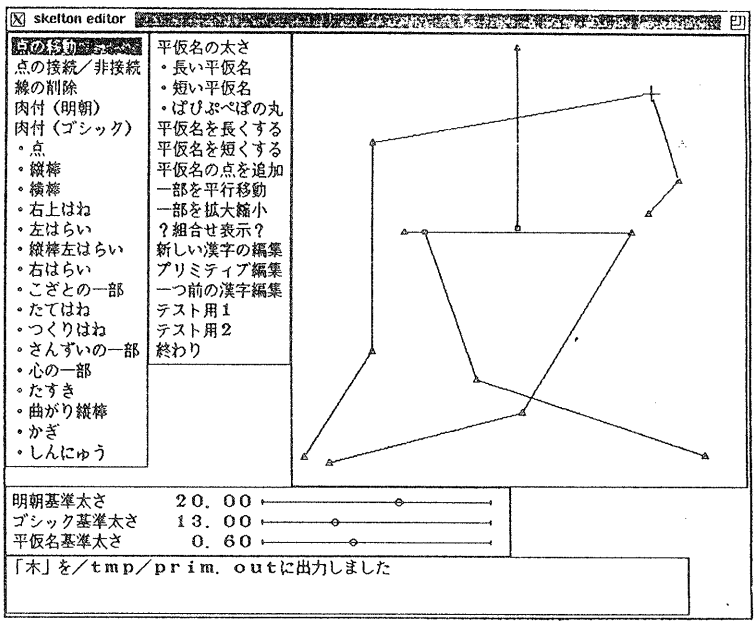

図 4: 編集画面「皮」(点の移動)

造体のメンバ変数である plist に注目した. これはユーザが自由に使ってよいメンバで, 最初は nil になっている. これに連想リストの形で色々なデータや関数を入れておくことにした. get-winprop はウィンドウ win の属性 key の値を取り出し, put-winprop はウィンドウ win の属性 key の値を value に変更する関数である. 描画の際に必要なグラフィックコンテクストや画面の再表示に必要なピックスマップ, 再表示関数などがみなこのアクセス関数を使って window の plist にいれてある.

```
(put-winprop win
  'exposure-handler
  (function redraw-win))
```

イベントに対応した関数を呼び出すために振り分け関数を使う.

```
(defun handle-exposure (win)
  (let ((func
          (get-winprop
            win
            'exposure-handler)))
    (and func (funcall func win))))
```

イベント発生ウィンドウは event-window という変数で参照できるので,

```
(handle-exposure event-window)
```

のようにイベントに対応した処理をさせることになる.

5.3 段階のある処理

1つの点を指定するだけで処理が済んでしまうもの (制御点の接続属性の反転など) についてはイベントに対応する関数を定義するだけで簡単に記述ができるが, 複数の点を指定しなければならない作業 (エレメントの追加など) では以下を考慮する必要がある.

- 1度定義を始めたら途中で別のメニューを選べないようにする. こうすると, イベントに対応する関数は1つでよいが使い勝手が悪い. 途中で終わらせるためには特殊なボタンを押すなどで対処する.
- 1段階終わるごとにイベントに対応する関数を変更していく. 途中で別のメニューを選べる利点があるが, 関数の制御が煩雑になる.

このエディタでは前者の手法を採用した. 中断ボタンも用意してあるが, 実際にはひとまず定義を最後までやってしまって, その後でその定義を削除するという手法が使われているようである.

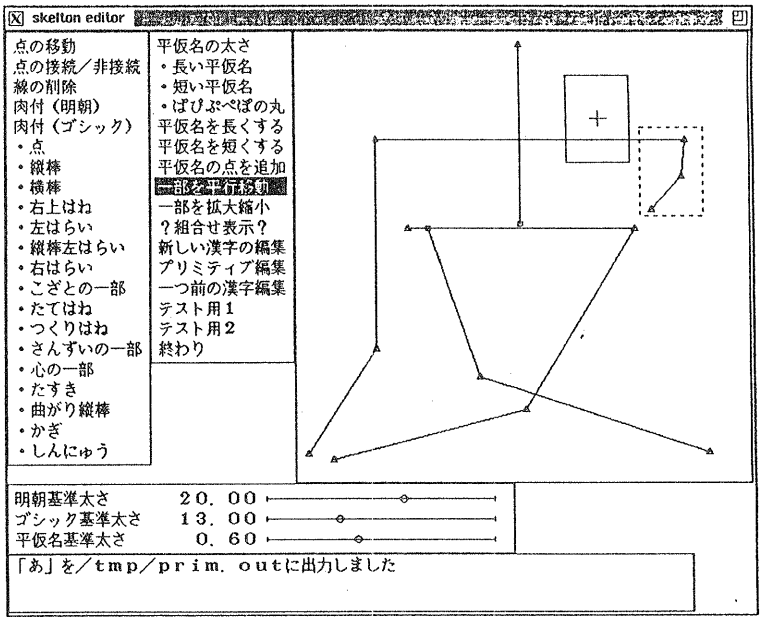

図 5: 編集画面「皮」(一部を平行移動)

5.4 ポインタを追いかける描画

ポインタの動きに追随して描画がおこなわれるようにすると, 状態がわかりやすくなる. 例えば, 部分の拡大縮小や移動では, 移動前が点線でしめされ, ポインタに合わせて移動先の候補を描画しながらしめしていく. このような場合, ポインタの位置が変わると次のような処理が必要になる.

1. 前に描画した位置の描画を消す.
2. 新しい位置に描画する.

このため, 前に描画した位置を記憶しておく必要がある. 変数名の衝突が起きないようにするためにも, 前の位置をいれておく変数は描画に関係のない関数からは見えない方がよい. UtiLispではパッケージを使うことで変数名をファイルごとに管理することができる.

5.5 試行錯誤による開発

新しい機能をつけくわえようとするとき, エディタのような対話的なアプリケーションでは実際に動かして挙動を観察しながら開発する方が効率がよい. そこで, テスト用のメニューをあらかじめ作っておき, そのウィンドウのイベント処理関数を定義しなおして検査した.

```
(put-winprop
  etc-1
  'button-press-handler
  #'(lambda (win code x y)
      (print (list win code x y))
      ...))
```

メニュー自体に新しい項目を付け加える関数を定義することもできるが, まだ対応はしていない.

5.6 データの内部表現

プリミティブはS式で表されるがエレメントの編集には制御点が何番目の点かという情報を多く使う. これはどちらかといえばリストよりも配列向きの作業である. 配列で点の座標をとっておいて出力時にS式に変換するという, エディタ内部表現を使う手法もあるが, これは採用しなかった. これは, 内部表現のフォーマットが忘れられがちであること, 他者が開発に関わってきた時にこの内部表現をきちんと伝えないといけないこと, などの難点があるからである. すでにS式という明白な形式が定まっているので, これをそのまま扱うようにした. 配列の変わりにリストを使うことによる速度的な問題はまだ起きていない.

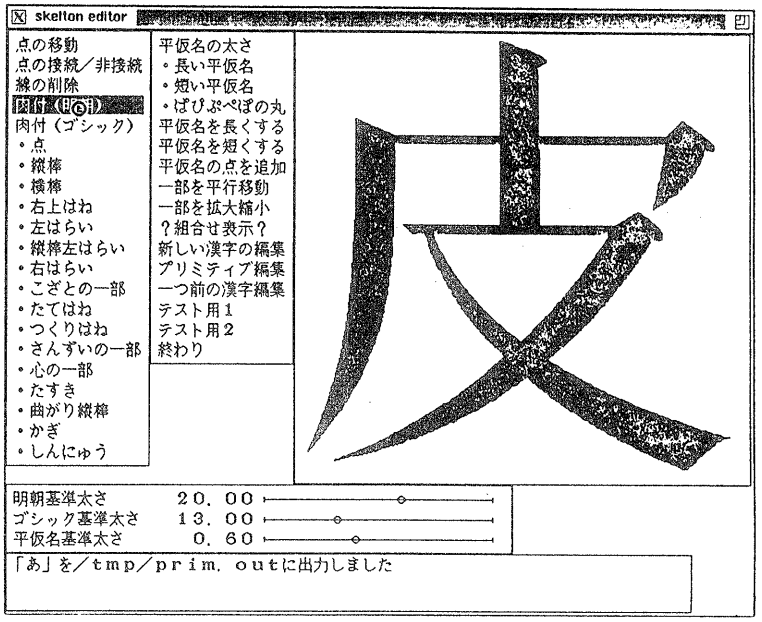

図 6: 編集画面「皮」(明朝体肉付け)

5.7 機能の限定

スクロールバーやカラー化など見栄えのする機能については対応していない. このような機能は素人目には格好よく見えるものだが実際にそれほど必要とされるものではないからである. それよりも, ボタンの押し / 放しを状況に応じてきめ細かく対応するようなことに気を配った.

6 おわりに

漢字スケルトンフォント作成支援ツールをUtiLispで作成したのでこれを紹介した. Lispの持つ柔軟なデータ構造や対話的な開発環境で効率的に開発を進めることができ, 使いやすいツールを短い期間で作り上げることができた. 新しい機能の追加の要望があってもすぐに対応ができるという利点をもっている.

なお, 本研究は文部省科学研究費補助金 (重点領域研究) (課題番号 03235103) の補助を受けている.

参考文献

[1] 田中ほか: 漢字スケルトンフォントの生成支援システム, 第 32 回プログラミング・シンポジウム報告集, pp. 1–8 (1991).

[2] 国西ほか: ベクトルフォント編集プログラムの機能と方式, 第 29 回情報処理学会全国大会論文集, pp. 1437–1438 (1984).

[3] Wada Laboratory: UtiLisp Manual Revision 2.0, (1988).

[4] 田中: SPARC の特徴を生かした UtiLisp/C の実現法, 情報処理学会論文誌, Vol.32, No.5, pp. 684–690 (1991).

[5] Texas Instruments Incorporated: CLX Common Lisp X Interface, (1989).