ユーザインタフェースのプログラミング

増井俊之
ソニーコンピュータサイエンス研究所
masui@csl.sony.co.jp
http://www.csl.sony.co.jp/person/masui/

インタフェースソフトウェア作成の困難さ

アプリケーションプログラム中の UI 部分の比率は 50% とか 80% とかよくいわれるが...

- リアルタイム性
- 複雑な要素技術
- アプリケーション部とインタフェース部の交錯
- プログラムの制御構造が複雑
- 柔軟性 / 頑強性
- ユーザ補助機能
- 評価 / 再設計の繰り返し

リアルタイム性

- 効率的実装
  - 素早い反応
  - 高速描画

複雑な要素技術

- グラフィック出力
- 画面配置
- 音声認識
- ジェスチャー認識

アプリケーション部とインタフェース部の交錯

- 多くのプログラムは計算部と入出力部が融合していて分離できず再利用できない

```
printf("Input your name: ");
scanf("%s", name);
s = calc(name);
printf("%s\n",s);
....
```

AP と IF の分離

- インタフェースプログラムを再利用できる
- ひとつのアプリケーションに異なるインタフェースを使える
  - ◇ 同じプログラムをウィンドウシステム上でも文字端末でも使える
  - ◇ 素人と玄人で異なるインタフェースを使える
  - ◇ 容易にカスタマイズできる
- テストモジュールやデモモジュールを簡単に接続できる

分離の困難さ

- 普通の手続き型言語では逐次実行文の字面を分離するのはむずかしい
- UI 部は AP 部の情報が必要になるし、逆も成り立つので疎結合にすることは本質的にむずかしい
  - ◇ (例) プルダウンメニューの中の使用不可能な項目を消す
    ← メニューを扱うプログラムがアプリケーションの実行状態を知っていなければならない

複雑な制御構造

- 複数の入出力装置を同時に扱う必要がある
  - マウスとキーボードを同時に使う
  - 複数のウィンドウ
  - 入力を受けつけながらアニメーションが動く
- インタフェース制御とアプリケーション本体制御の交錯
  - 計算に時間がかかるとき入力が受け付けられないと困る
  - 計算機の質問にユーザが答える (システム主導)/ ユーザが主体で指示を出す (ユーザ主導)/ それらの融合
- 例外処理が多い
  - 時間切れ
  - アボート
  - ヘルプ
  - UNDO

柔軟性 / 頑強性

- 操作の間違いに対応
  - ◇ 操作間違いが致命傷にならないようにするため前の状態を記憶しておくなど冗長な資源を用意
  - ◇ あらゆる入力、条件において異常動作がおこらないようにする
    - – テスト手法が重要
  - ◇ 操作を間違えにくいようにする
    - – モードの廃止など

ユーザ補助機能

- ヘルプ、ガイダンス、マニュアル
- ユーザ適応
- 操作予測

評価 / 再設計の繰り返し

- 最初に決めた仕様どおり動けば完成というわけにいかない
  - ◇ 悪いインタフェースを発見して何度も試行錯誤 (作りなおし) が必要
- 評価に時間と労力がかかる
- ラピッドプロトタイピングと変更の容易さが必要
- 簡易インタプリタ言語が有効

インタフェースソフトウェア作成に適用可能な技術

- オブジェクト指向技術
  - ◇ プログラムのモジュール化
  - ◇ 部品の再利用 (e.g. ウィンドウ、メニュー)
  - ◇ 並行オブジェクト
- 各種言語の使用
  - ◇ オブジェクト指向高級言語
  - ◇ 簡易インタプリタ言語
    - – 試行錯誤のサイクルを高速にするのに有効
  - ◇ 視覚言語
    - – 画面設計や状態遷移の設計などに有効
  - ◇ 並行処理言語
    - – 同時に発生する動作の処理に有効

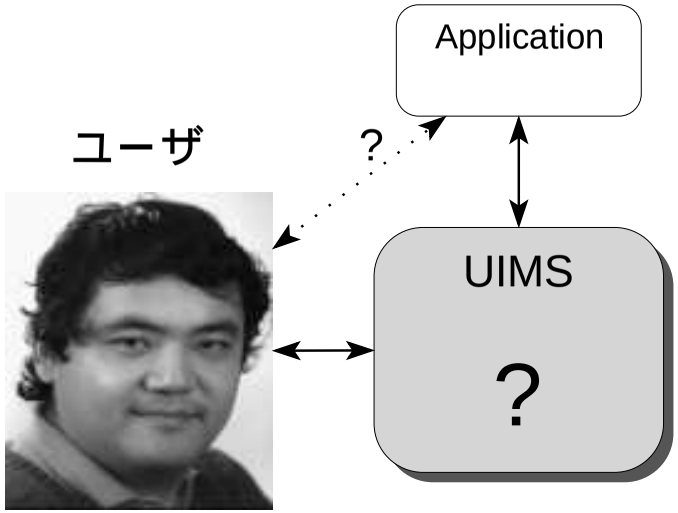
インタフェースソフトウェア作成に適用可能な技術 (Cont'd)

- ソフトウェア開発環境
  - ◇ プログラム編集、デバッグ、保守、マニュアル作成、 etc.
- ユーザインタフェース専用システム
  - ◇ 制約解決システム
  - ◇ 例示プログラミングシステム
  - ◇ 適応 / 学習システム

解決法 1: UIMS

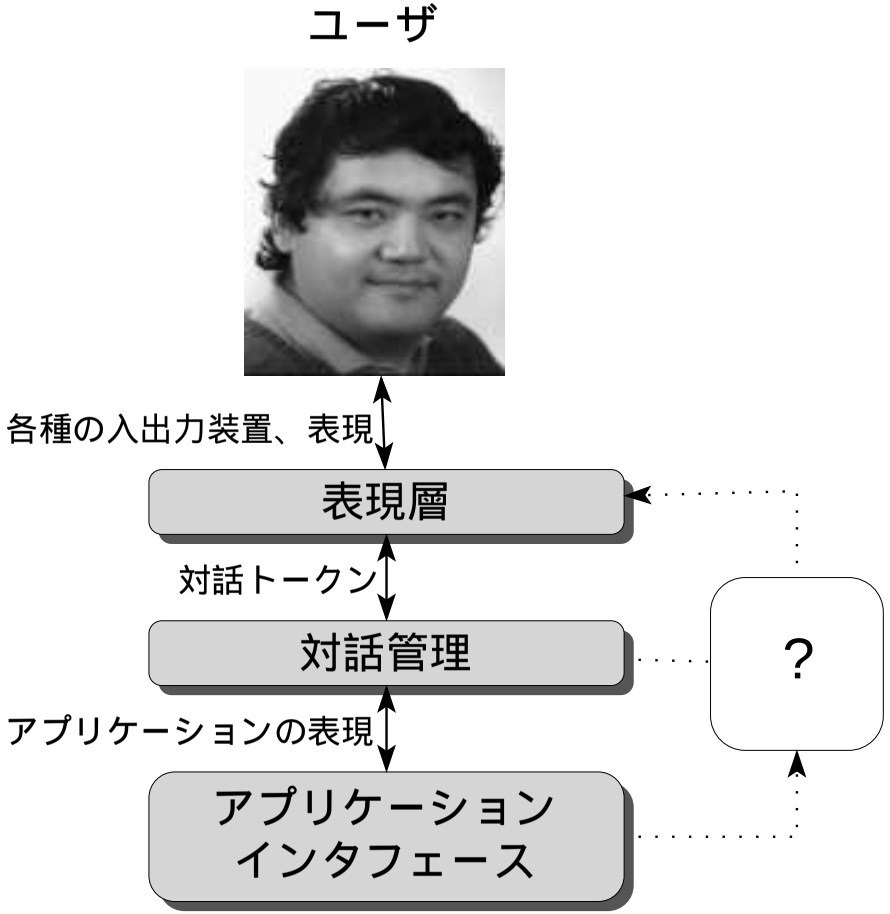
UIMS = User Interface Management System

- アプリケーションとインタフェースの分離を支援
- 複雑な部分は全て UIMS 中にもつ



Seeheim モデル

- UIMS の有名な (しかし古い) モデル
- インタフェースとアプリケーションを抽象化して分離



- 現存のインタフェースにはほとんど適用不可能
- 意味フィードバック (アプリケーション内の情報がインタフェースに直接反映される) が実現できない

UIMS の問題点

- あらゆるインタフェースパターンを用意するのは不可能
  - ◇ インタフェースの方式に制限がついてしまう
- にもかかわらずシステムが巨大になりがちである
- 完全に分離を支援することは不可能
  - ◇ アプリケーションに依存したインタフェースもある

→ 近年あまり流行っていない

解決法 2: UI 記述言語

- インタフェース記述専用言語を設計し、インタフェースの複雑なところはそれで記述する
- UIMS は専用の UI 記述言語を持っていることが多い
- インタフェース言語の例
  - HyperCard
  - Squeak[1]: CSP ライクな並列言語で UI を記述
  - ERL[4]: Sassafras UIMS のイベント言語

UI 記述言語の問題点

- 特殊な専用言語を覚えなければならない
- UI 記述言語がアプリケーション記述言語と異なる場合それらの間の整合性が問題になる
- UI 記述言語でアプリケーション自体も記述する場合、汎用言語の全機能が UI 言語に必要になり、言語仕様が巨大化する
- UI 言語に必要な UI 機能を絞り込むことはできない
- 機能を限定すれば成功することもある (e.g. HyperCard, Lingo)

解決法 3: GUI ツールキット

- インタフェース部品をライブラリ化して再利用
- イベント駆動 (マウス、タイマ、 etc.)
- 制御の主体がツールキット側にあり、イベント発生により「コールバック関数」が呼び出される

各種のグラフィックツールキット

- OpenLook, Motif, XView, Athena Widget, HP Widget, NEXTSTEP, XForms, ...
- MacApp
- Windows Foundation Class
- Java AWT
- Tcl/Tk (STK, PerlTk, PythonTk, etc.)
- Garnet, Amulet
  - Carnegie Mellon University (Brad Myers)
  - 独自オブジェクトシステム、制約解決
- subArctic
  - Georgia Tech (Scott Hudson)
  - Java ベース、制約解決

オブジェクト指向ツールキット

- 「ウィンドウ」「テキスト枠」などのインタフェース部品は独立して動き、それぞれが個有のデータや手続を持っているため、オブジェクト指向言語での実装に都合が良い

  例: NeXTstep のプログラム

```
main(){
    NXApp = [MyApp new];
    window = [NXApp mainWindow];
    mask = [window eventMask];
    [NXApp run];
    ...
```

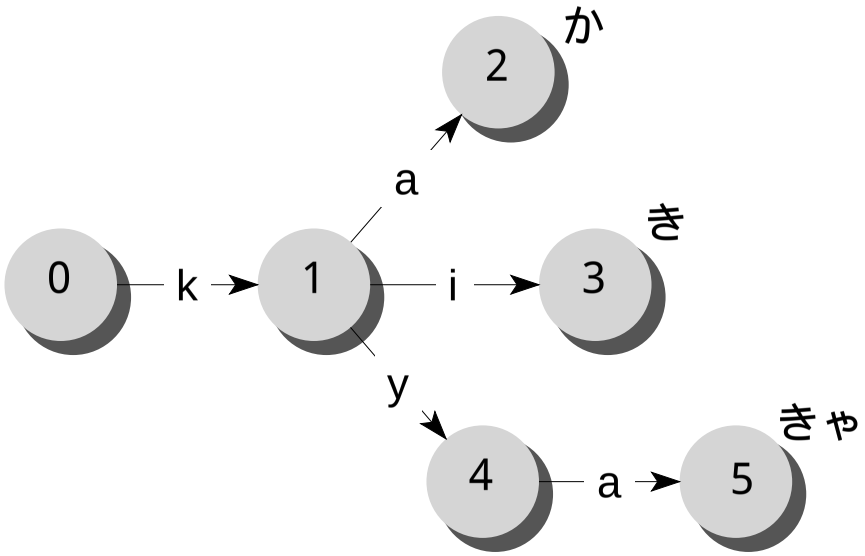
- ウィンドウ、テキスト枠などよりも粒度の細かいもの (文字など) までオブジェクトとするツールキットもある。

UIツールキットの問題点

- インタラクション手法が決まってしまう
  (インタフェースの統一がとれる点は利点)
- アプリケーションが「コールバック関数」としてインタフェースライブラリに呼び出されるという型式のことが多く、制御の自由が失われる
- 必要なライブラリの数が多い
- 拡張が不可能であるか難しい
- システムや言語が制限される

解決策 4: 状態遷移機械

- ユーザ入力などの事象の発生によるプログラムの実行状態の変化を記述することにより UI の記述を楽にする
- 例: ローマ字かな変換の状態遷移



状態遷移機械の記述例

- 状態遷移の記述 (Flex(後述) によるもの)

```
ka      { printf("か"); }
ki      { printf("き"); }
kya     { printf("きゃ"); }
```

- 普通に (?) プログラミングした場合

```
c = getc();
if(c == 'k'){
   c = getc();
   switch(c){
     case 'a': printf("か"); break;
     case 'i': printf("き"); break;
     case 'y':
        c = getc();
        if(c == 'a') ...
```

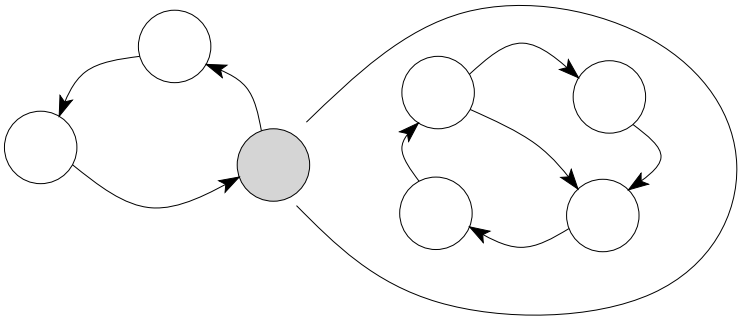
状態遷移機械の問題点

- インタフェースが複雑になると状態の数が非常に多くなる
  - ◇ ひとつの画面上で表現しきれない
- モードレスインタフェースを実現しにくい
- 状態遷移記述だけではプログラミングが楽にならない

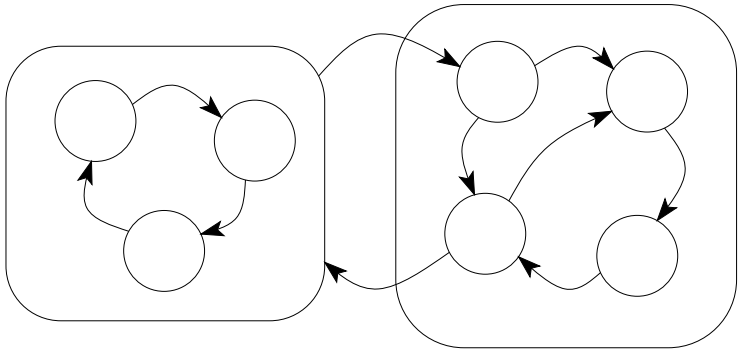
拡張状態遷移機械

単純な状態遷移機械を拡張したモデル

- 階層的状態遷移機械



- StateChart[3]



  - ◇ 並列性を記述できる
  - ◇ 複数の状態のモジュール化

- ペトリネット
  - 並列性を記述できる
  - 高機能すぎる?

解決策 5: 並列言語

並列処理は UI プログラミングに非常に重要

- 複数の入出力装置を使う
  - ◇ マウス、キーボード、 etc.
- 複数の入出力を同時に行なう
  - ◇ 例: 車の運転 (ハンドルを回しながらアクセルを踏む)
- モジュール分割
  - ◇ アプリケーション部とインタフェース部を並列に動かす
- ツールキット部品を並列に動かす

  → モードレスインタフェース
- 複数ユーザでも大丈夫

並列言語の問題点

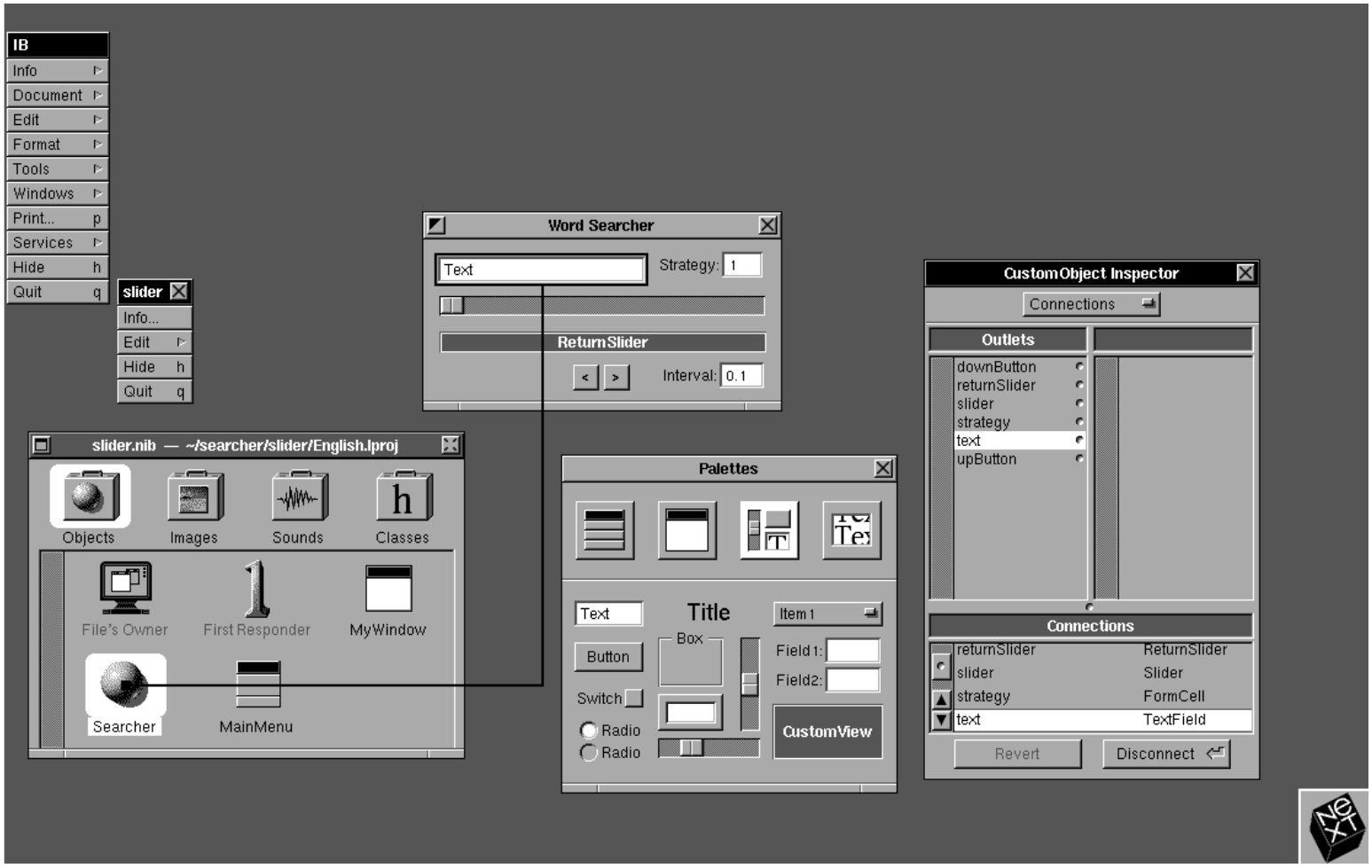
- 特殊な言語が必要
- アプリケーション言語との整合性
- デバッグがむずかしい
- アプリケーションプログラマが並列プログラミングに慣れていない

解決法 6: インターフェースビルダ

NeXT のものが最初だが最近は各社が製品を出している

- グラフィックツールキットをグラフィックエディタで対話的に配置
- アプリケーションプログラム (の骨格) の自動生成
- コンパイルせずに操作を実験

Interface Builder (NeXT)



インターフェースビルダの問題点

- システムやツールキットに依存する
- 対話方式が制限される
- グラフィック UI にしか使えない
- システムが複雑

but...

ウィンドウシステム上の GUI 開発においては現状では最善と考えられるため、各種の高機能インタフェースビルダが今後開発されてくるであろう

解決法 7: 制約システム

- オブジェクト間の制約を宣言的に記述し、システムが自動的にそれを解決するようなシステム
- 適用分野
  - ◇ グラフィック図形間の制約
    - 「ある図形が別のふたつの図形のちょうど真中」のように指定しておくと、それぞれを移動させたとき残りも追随する
  - ◇ スプレッドシート
    - 制約が破られると自動的に再計算が行なわれる
    - いわゆるスプレッドシート以外のアプリケーションにも適用可能

制約システムの問題点

- 適用範囲が狭い (PDA に役にたつか??)
- 制約が複雑なとき解くのに時間がかかる
  対話的に使えない場合がある
- 簡単な制約しか解くことができない

解決法 8: 例示プログラミングシステム

- 自分でプログラムを書かず、操作の例をシステムに与えることによりプログラムを作成するシステム
- PBE (Programming by Example) とも呼ばれ、 UI 以外の分野でも使われている。 (例: QBE (Query by Example) – 例示によりデータベースの検索を行なう)
- 具体的な例を一般化したり制御構造を推定したりする

例示システムの問題点

- 単純なプログラミングにしか使えない
- システムが予想可能な動きに限られる
- UI プログラムの開発者用というよりも一般ユーザ向け
  マクロ定義、カスタマイズの一種と考える

ではどうすればよいか?

完璧な方法は無く、どの手法も問題は多いが、インタフェースソフトウェア技術は確実に進歩しているので、現状で最も有効なテクニック / ツールの組みあわせを選択するのがよい。
どのツールを使うにしても、以下のような注意点を念頭におくのがよいであろう。

- モジュール分割を意識する
- 並行処理の手法を使う
- 状態遷移記述などは普通のプログラミングと別の記法を使う
- インタラクションを既定せず変更を容易にする
- インタプリタを使って開発 / 実験サイクルを短くする
- 分野に応じた便利なツールを捜す (制約システム、 etc.)

From scratch, 小規模システムの場合

- 便利そうな小ツールを捜す
- スタンドアロンのコンパクトなツールを使う
  UIMS や大規模ツールキットを使わない
- 状態遷移の簡単な記述
- 何らかの並行処理を使う
  - ◇ 並列言語
  - ◇ コルーチン
  - ◇ スケジューラ

From scratch, 大規模システムの場合

- 大規模で有用なシステムの使用を検討する
- ツールキットを活用する
- 場合によってはツールキットから自作する
- 並列言語を使う

ウィンドウシステムの場合

- インタフェースビルダのあるツールキットを使う
- 使いやすいツールキットを選択する
- 高度なグラフィックシステムを選択する

現状では目的と環境に応じて以下のような組みあわせが適当 (開発が楽)

- NeXTstep + InterfaceBuilder
  - ◇ オブジェクト指向グラフィックツールキット, Display PostScript, 元祖インタフェースビルダ
- X11 + XForms
  - ◇ 簡易インタフェースビルダ (XForms), 標準ウィンドウシステム (X11)
- SGI/Windows + OpenGL(3D グラフィクス) + GLUT(OpenGL 拡張)
- Java AWT

- X11/Windows + Tk(簡易ツールキット) + Tcl/Scheme/Perl/Python
- Garnet
  - X11 + Common Lisp Object System
  - 各種のインタフェースのアイデアを統合 (オブジェクト指向、制約、etc.)
  - 新版は Amulet (X11 + C++)
- subArctic
  - Java のツールキット
  - 制約機構

プロトタイプシステムをとりあえず動かす場合

- 目的にあった簡易システムがないか調べる
  - Visual Basic, Delphy, etc.
  - Director などのオーサリングシステム
  - Prograph、 Java Studio などの Visual Language
- インタプリタ言語で間にあわないか調べる
  - CLOS + MacApp
  - CLX (X11 + CommonLisp)
  - Tk (X11 + Tcl/Schem/Perl/Python/etc.)
- どうしようもない場合だけプログラミングする

ツールの実例 1: Flex[5][7]

- Flex = 状態遷移コンパイラ
- 状態遷移を正規表現で記述したものをコンパクトな遷移表に変換
- 効率の良い C コードを生成

```
%%
ka      { printf("か"); }
ki      { printf("き"); }
kya     { printf("きゃ"); }
...
```

→ 状態遷移関数**zztrans(**入力文字, 遷移機械番号**)** と遷移表を生成

- 適用領域
  - ◇ UI の状態遷移記述
  - ◇ エスケープシーケンス解析
  - ◇ ローマ字かな変換

Flex 適用例

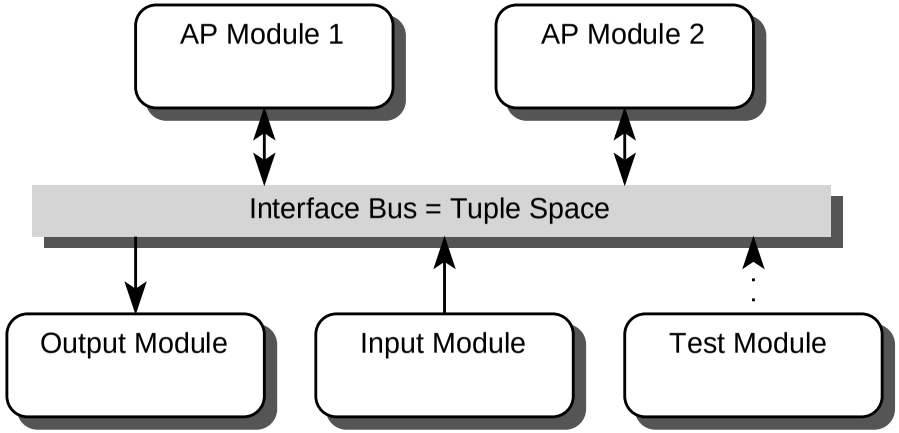
```
%%
a { printf("あ"); }
ba { printf("ば"); }
be { printf("べ"); }
bi { printf("び"); }
bo { printf("ぼ"); }
bu { printf("ぶ"); }
bya { printf("びゃ"); }
bye { printf("びぇ"); }
byi { printf("びぃ"); }
...
zye { printf("じぇ"); }
zyi { printf("じぇ");}
zyo { printf("じょ");}
zyu { printf("じゅ");}
```

↓↓↓

```
static short zznstate[] = {
 25, 92,  2, 31,  4, 90, 13,  7, 15, 87, 11,  5,  6,  8, 81, 10,
  1,  3, 18, 75, 29,  9, 22, 23, 82, 69, 99,175, 83, 65,100,174,
 84, 59,101,176,177, 85, 52,102, 88,  0, 86, 63, 89, 17, 28, 12,
...
125,  0,113,151,126,  0,  0,116,152,127,  0,  0,117,153,128, 16,
134,139,119,  0,135,140,120,  0,136,141,121,  0,  0,137,142,122,
  0,  0,138,143,123 } ;
zztrans(c,n) int c; int n;
{
int *state;
int index;
  state = &(zzstms[n].zzstate);
if(zzdeftrans[*state])
*state = zzdeftrans[*state];
else{
index = zznindex[*state] + zzctype[c];
...
```

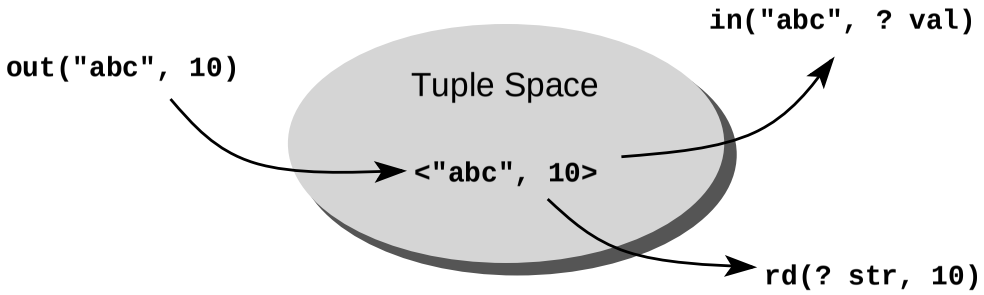
ツールの実例 2: Linda[2][5]

- Linda = 並列言語プリミティブ
- 共有空間を使った同期 / 非同期通信
- ライブラリの形で既存言語に融合可能
- モジュールの分離が可能



Linda モデル

- タプル (データ組) をタプル空間 (共有空間) でやりとりする
- ４種類のプリミティブ
  - **out** – タプルをタプル空間に書き出す。
  - **in** – タプルをタプル空間から読む。タプルは消去する。
  - **rd** – タプルをタプル空間から読む。タプルは消去しない。
  - **eval** – プロセスを生成する。

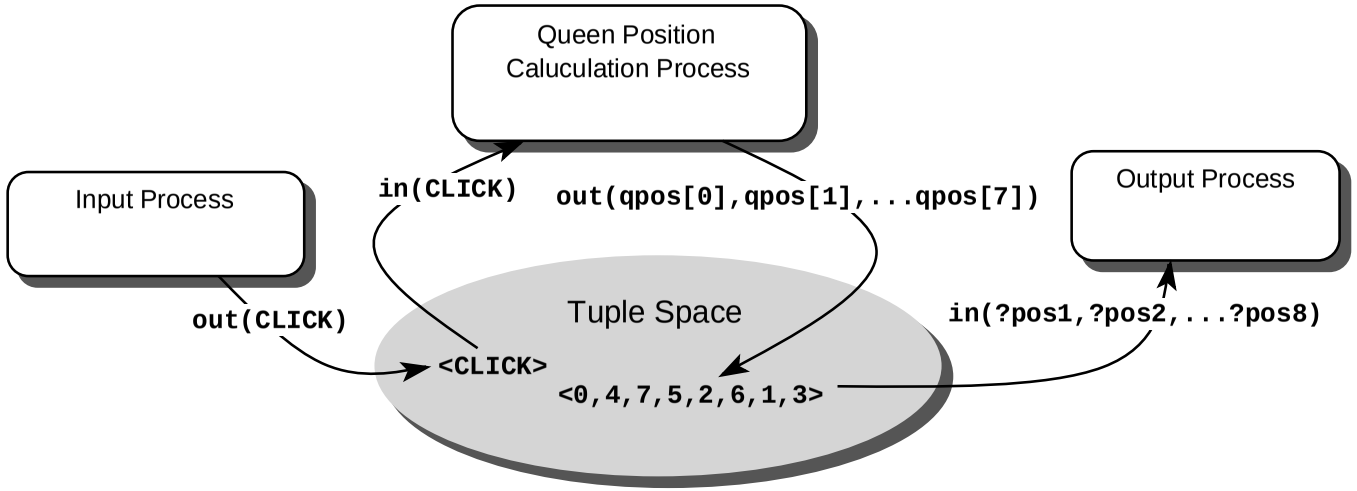


8-Queen

```
...
int queens = 8;
main(argc, argv) char **argv;
{
    int i;
    for(i=0;i<queens;i++){ col[i] = qpos[i] = 0; }
    for(i= -queens;i<queens;i++){ up[i] = down[i] = 0; }
    extend(0);
}
extend(n)
{
    int c;
    for(c=0;c<queens;c++){
        if(!col[c] && !up[n+c] && !down[n-c]){
            qpos[n] = c;
            if(n+1 >= queens)
                printqueens();
            else {
                col[c] = up[n+c] = down[n-c] = 1;
                extend(n+1);
                col[c] = up[n+c] = down[n-c] = 0;
            }
        }
    }
}
```

- 再帰の底 **`printqueens()`** が呼ばれている
- 「Queen をひとつ計算して印刷する関数」を作りにくい
  マウスクリックの度に解を表示するインタフェースを作りにくい

Linda による解法


Queen Position
Caluculation Process
Input Process
Output Process
in(CLICK)
out(qpos[0],qpos[1],...qpos[7])
out(CLICK)
Tuple Space
in(?pos1,?pos2,...?pos8)
<CLICK>
<0,4,7,5,2,6,1,3>

端末インタフェースをもつ 8-Queen プログラム

```
<定義>
main()
{
    <初期化>
    eval(extend(0));  //  8-Queen計算プロセス起動
    for(;;){
        //  入力は存在せず、解が見つかる度に印刷される。
        out(CLICK);
        in(?p1,?p2,?p3,?p4,?p5,?p6,?p7,?p8);
        printf("%d  %d  %d  %d  %d  %d  %d  %d¥n",
                    p1,p2,p3,p4,p5,p6,p7,p8);
    }
}
```

ウィンドウインタフェースをもつ 8-Queen プログラム

```
<定義>
int  click();
ExecBox  *e;
main()
{
<ツールキット初期化>
    eval(extend(0));  //  8-Queen計算プロセス起動
    //  「実行ボックス」を生成し、マウスでクリック
    //  される度にclick()が起動されるようにする。
    //  プロセスは自動的に生成される。
    e  =  new  ExecBox(posx,posy,w,h,click);
    for(;;){
        in(?p1,?p2,?p3,?p4,?p5,?p6,?p7,?p8);
<ウィンドウにチェス盤を表示し、p1..p8の
          対応する位置にクイーンを表示する。>
    }
}
click()   //  マウスクリックにより起動される
{
    out(CLICK);
}
```

参考文献

1. Luca Cardelli and Rob Pike. Squeak: a language for communicating with mice. *Proceedings of SIGGRAPH*, Vol. 19, No. 3, pp. 199–204, July 1985.
2. David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, pp. 80–112, January 1985.
3. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, Vol. 8, pp. 231–274, 1987.
4. Ralph D. Hill. Supporting concurrency, communication, and syncronization in human-computer interaction: The sassafras uims. *ACM Transactions on Graphics*, pp. 179–210, July 1986.
5. Toshiyuki Masui. User interface specifacation based on parallel and sequential execution specification. In *USENIX'91 Conference Proceedings*, pp. 117–125. USENIX' Conference Proceedings, January 1991.
6. Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Marchal Pilippe, Ed Pervin, and John A. Kolojejchick. The garnet toolkit reference manuals: Support for highly-interactive, graphical user interface in lisp. Technical Report CMU-CS-89-196, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1989.
7. 増井俊之. 効率の良いトライ / 状態遷移機械の構成方式. 情報処理学会 プログラミング – 言語・基礎・実践 – 研究会研究報告 94-PRG-15, Vol. 94, No. 7, pp. 73–80, January 1994.

有用 URL (再掲)

- インタフェース関連リンク集 http://www.csl.sony.co.jp/person/masui/Articles/HCIBookmark.html
- ACM SIGCHI (http://www.acm.org/sigchi/)
- Jacob Nielsen(ユーザビリティ工学 /Web デザイン研究者) のページ (http://www.useit.com/)
- HCI Bibliography (http://www.hcibib.org/)
- 講義サポートページ (http://www.csl.sony.co.jp/person/masui/KeioHI/)