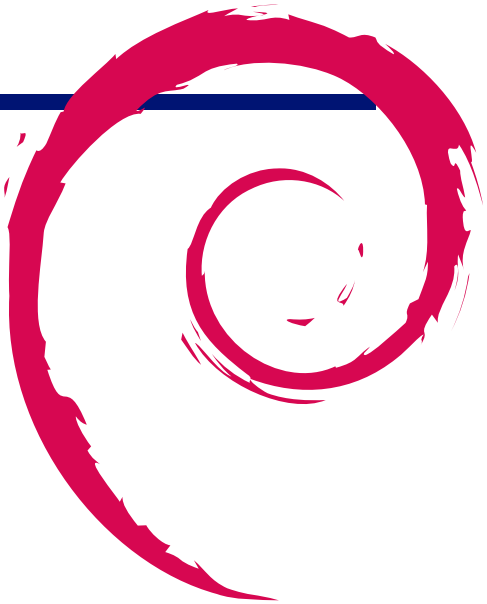
東京エリア デビアン 勉強会

Debian勉強会幹事 上川純一

2010 年 2 月 20,21 日



1 Introduction

上川 純一

今月の Debian 勉強会へようこそ。これから Debian の世界にあしを踏み入れるという方も、すでにどっぷりとつかっているという方も、月に一回 Debian について語りませんか?

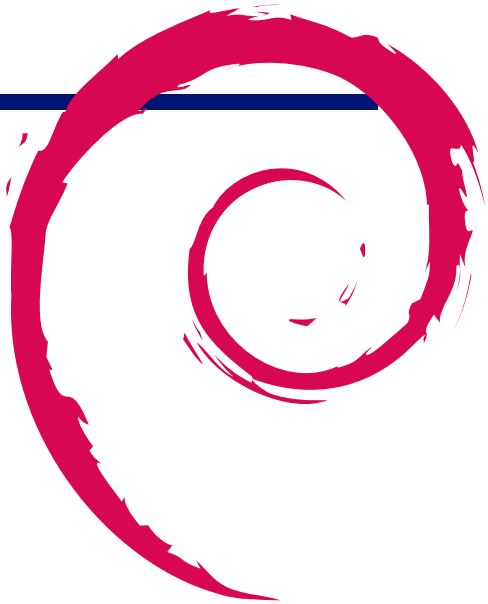
Debian 勉強会の目的は下記です。

- Debian Developer (開発者) の育成。
- 日本語での「開発に関する情報」を整理してまとめ、アップデートする。
- 場の提供。
  - 普段ばらばらな場所にいる人々が face-to-face で出会える場を提供する。
  - Debian のためになることを語る場を提供する。
  - Debian について語る場を提供する。

Debian の勉強会ということで究極的には参加者全員が Debian Package をがりがりと作るスーパーハッカーになった姿を妄想しています。情報の共有・活用を通して Debian の今後の能動的な展開への土台として、「場」としての空間を提供するのが目的です。

デビアン勉強会

目次



2 事前課題

上川 純一

今回の事前課題は以下です:

1. Debian について現在知っていることを教えてください。
2. Debian について知りたいとおもっていることを教えてください。
3. Debian を使う理由を挙げてください。

この課題に対して提出いただいた内容は以下です。

2.1 キタハラ

1. 創始者の Ian さんとその妻 Debra さんの名前から命名された。
2. いっぱいありすぎて、回答欄におさまらない。
3. 一番の理由は、自分があまのじゃくだからかなぁ？

2.2 emasaka

3. とにかくたくさんのパッケージから構成を選んで、あるいは必要に応じてパッケージを追加して、なおかつある程度安心して、用途に合ったシステムを組める点。

2.3 henrich

1. dis られながらもがんばる健気なプロジェクトです。
2. プロジェクトリーダーの活動! 一体何してるの? ?
3.「縁があったから」です :)

2.4 mkouhei

3. 複数のアーキテクチャのマシンを、その違いを気にせずに利用できるパッケージシステムが便利すぎるためです。使い始めたきっかけでもあります。うちで使っているのだけでも 5 種類です。

2.5 吉野 (yy_y_ja_jp)

3. DFSG があるから．&パッケージングシステムが優れているから．ですね．

2.6 日比野

3. 学生時代に理想のパッケージ管理システムに出会ったと思って以来、ずっと使ってます。

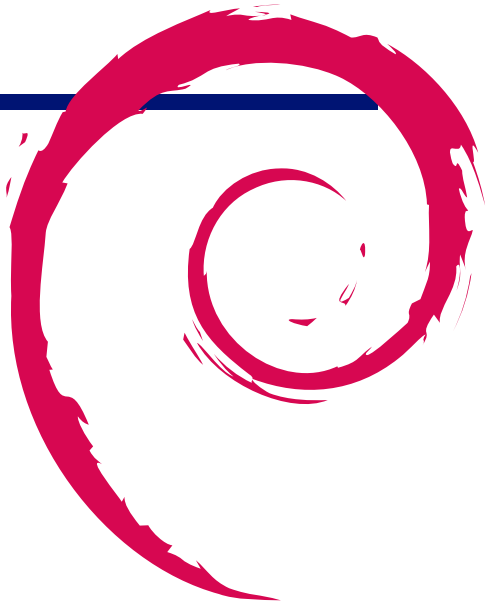
2.7 なかおけいすけ

3. apt の存在に気づいた時、他のディストロを使う理由が無くなったなぁと感じたことを覚えています。

   次に、 stable release が保守的なことです。 potato から debian を使い始めたのでこういう印象があるのかもしれませんが、長期間使い続けることが前提の用途が多いので、 stable は変わらないで欲しいのです。 debian の stable は、その点都合が良い。

   あと、 X-Window をむやみやたらに入れたがらないところも良いです。最近のディストロは、やけに X を入れたがるような印象があります。

   最後に、 debian が、ある日突然有料になったり、無くなったりしそうにないディストロであるということです。 Debian Social Contract がある限り debian は debian でありつづけるのかなと思っています。

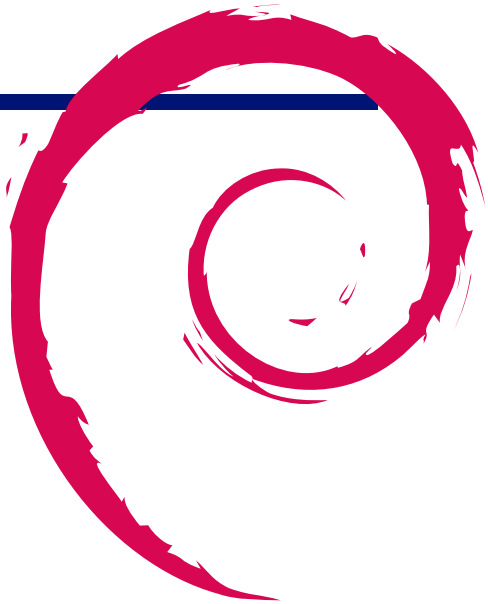


3 最近の Debian 関連のミーティング報告

上川純一

3.1 東京エリア Debian 勉強会 60 回目報告

2010 年の 3 月に次期安定版である コードネーム ”squeeze” がリリースフリーズに入る予定です。 これに合わせて世界各地で Debian Bug Squashing Party が行われます。 前回の 東京エリア Debian 勉強会は 勉強会をかねて、この Bug Squashing Party を東京大学先端科学技術研究センターに会場をお借りして実施しました。



4 Debian の紹介

やまねひでき

4.1 Debian とは何か

「 Debian とは一体何ですか? [*1]」には以下のように書かれています。

> Debian Project は、フリーなオペレーティングシステムを作成するために連携した個人の集団です。 我々が作成したこのオペレーティングシステムは Debian GNU/Linux もしくはもっと短かく簡単に Debian と呼ばれています。

4.2 Debian の特徴

今だと Windows/MacOSX 以外の「 いわゆるフリーな OS」はいくつかあります。では、他の OS / ディストリビューションと Debian の違い、その特徴を語るキーワードとは何でしょうか? 私は「 Universal OS」「 フリー」「 ボランティア」の三つを挙げます。順を追って説明します。

4.2.1 Universal OS

これが Debian が目指すものです。その意味するところは「 あらゆるマシンで動くフリーなソフトウェアによる誰もが使える OS」です。単に PC で動くだけではなく、最近は廃れてきましたが UNIX ワークステーションや汎用機、組み込み用機器、モバイル端末、ゲーム機…あらゆるマシンで動作することを目指しています。そのため多数の CPU アーキテクチャをサポートしているのが特徴です。サポートする／した／しようとしているアーキテクチャは以下があります。

- i386（ 通常の PC）
- amd64（ 最近の 64bit CPU）
- ia64（ 流行らない Intel の 64bit CPU。 Itanium など）
- mips/mipsel
- arm/armel（ シャープの Netwalker やモバイル端末がこれ）
- alpha
- hppa（ HP のワークステーション）
- sparc（ Sun）
- powerpc
- m68k（ 昔の Macintosh や Amiga など）
- s390（ 汎用機です）
- sh（ 日立の）
- avr32

また、その動作の核となるカーネルも Linux だけではなく他のカーネルに取り替えても動作することを目指しています。この移植版としては

- Hurd（ 永遠の開発版? ）
- kfreeBSD (i386, amd64)[*2]

[*1] `http://www.debian.org/intro/about`
[*2] NetBSD, OpenBSD は途中で作業する人の気力が尽きているようです。

があります。[*3]

単に動作する機器／カーネルが多いだけではなく、その上のユーザランドのソフトも豊富で、パッケージ化されており導入が容易になっています。現在リリースされている Debian 5.0 コードネーム「Lenny」ではその数は 25,000 パッケージを越え、その数はさらに増えつづけています。Linux で使えるソフトウェアを探す場合、大抵は既に Debian のパッケージとして提供されているので気軽に試すことができるでしょう。

それから Debian で利用可能な言語は多種に渡ります。それは自然言語（英語、日本語など）でもあり、計算機言語という意味でもあります[*4]。巷ではマイナーと呼ばれるような言語であっても「Universal OS」を目指す Debian は積極的に取り込んでいます。例えば、ブータン公用語「ゾンカ語」をサポートする DzongkhaLinux は Debian をベースに開発され、その成果は Debian に取り込まれています[*5]。

4.2.2 フリー

Debian の考える「フリー」は単に無料に止まらず、Debian フリーソフトウェアガイドライン (DFSG) という形でまとまっており、これが元になって「オープンソース」が生まれました。この点が担保される、この考えを皆が共有することでさらに豊かなソフトウェア／コンテンツ／社会が生まれています。このフリーというのは考えてみると中々奥深いものがありますので、ぜひ DFSG には一度目を通した上で Debian の考えるフリーという意味について Debian Developer の方などと話をしてみてください。

4.2.3 ボランティア

最後のキーワードです。Debian はその開発や財政基盤を会社や財団に持たない極めて稀有な開発集団です。大抵の有名ディストリビューションが企業をバックに開発をしていたり財団を持ってそのいたりする[*6]のですが、Debian 自体は財団や企業を持ちません[*7]。ボランティアが世界中でインターネットを介して開発するという状態が 10 年以上も続けられており、その規模は 1000 人を優に越えています。

4.3 誰が Debian を使っているの?

では、実際に誰が Debian を使っているのでしょうか?「仕事で使うなら Red Hat Enterprise Linux かそのクローンの CentOS が普通だよね〜」などと言い切っている人はいませんか? 実は、世の中に Debian で実際のビジネスを回している企業は山のようにあります。その中にはあなたが知っている企業もあるはずです。また、開発に愛用しているという方も少なくありません。あなたが使っているソフト／サービスは実は Debian が動いている／ベースになっている…かも知れませんよ。

4.4 最後に

簡単ではありますが、Debian の紹介をさせて頂きました。これも何かの縁だし Debian を使ってみてもいいかな、と多少でも思っていただければ幸いです。

4.5 Debian フリーソフトウェアガイドライン

Debian フリーソフトウェアガイドライン全文[*8]を掲載します。

[*3] 残念ながら Plan9 はありませんが、その上で動くツール類は移植されています。

[*4] 計算機言語の話は後で別の方が滔々としてくれるでしょう :-)

[*5] これは商用 OS では「採算にあわない」のでサポートが遅れがちになる少数言語／民族にとっての希望の現れと言えるでしょう

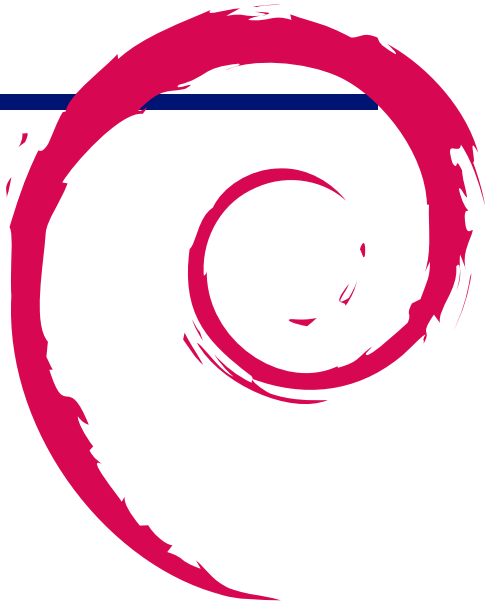
[*6] Fedora ← Red Hat, openSUSE ← Novell, Ubuntu ← Canonical, OpenOffice.org ← Oracle (Sun), Firefox ← Mozilla Foundation/Corporation など

[*7] 寄付などのために Software Public Interest という別法人がいますが、これは Debian だけではなく PostreSQL なども支援しています

[*8] `http://www.debian.org/social_contract#guidelines`

1.「 自由な再配布」…Debian システムを構成するソフトウェアのライセンスは、そのソフトウェアを、複数の異なる提供元から配布されているプログラムを集めたソフトウェア ディストリビューションの一部として、誰かが販売したり無料配布したりすることを 制限してはいけません。また、ライセンスはそのような販売に対して 使用料やその他の手数料を要求してはいけません。
2.「 ソースコード」…プログラムにはソースコードが含まれていなければならず、 かつ実行形式での配布に加えてソースコードでの配布をも 許可していなければなりません。
3.「 派生ソフトウェア」…ライセンスは、ソフトウェアの修正や派生ソフトウェアの作成、並びにそれら をオリジナルソフトウェアのライセンスと同じ条件の下で配布することを認め ていなけばいけません。
4.「 原作者によるソースコードの整合性維持」…ライセンスは、プログラムを構築時に変更する目的でパッチファイル をソースコードとともに配布することを容認している場合に限り、 ソースコードを修正済の形式で配布することを制限することができます。 この場合、そのライセンスは修正済のソースコードから構築されたソフトウェアの 配布を明示的に許可していなければなりません。 またライセンスは派生ソフトウェアにオリジナルソフトウェアと異なる名前 を付けること、あるいは異なるバージョン番号を付けることを要求できます (これは妥協案です。 Debian グループは全ての作者に、ファイル、 ソース、バイナリについての変更を制限しないよう奨めています)。
5.「 すべての個人、団体の平等」…ライセンスは、すべての個人や団体を差別してはなりません。
6.「 目標分野の平等」…ライセンスは、人々が特定の目標分野でプログラムを利用することを 制限してはいけません。たとえば、商用利用や、遺伝学の研究での プログラムの使用を制限していてはいけません。
7.「 ライセンスの配布」…プログラムに付随する権利は、プログラムが再配布された すべての人々 に対して、追加ライセンスの履行を必要とすることなく、 適用されなければなりません。
8.「 ライセンスは Debian に限定されない」…プログラムに付随する権利は、プログラムが Debian システムの 一部であるかどうかに左右されてはいけません。 プログラムが Debian から取り出され Debian とは別に使用 または配布されるとしても、その他の点でそのプログラムの ライセンス条項を満たしているならば、プログラムが再配布された すべての当事者は Debian システムにおいて付与されたのと 同じ権利を与えられなければなりません。
9.「 ライセンスは他のソフトウェアを侵害しない」…ライセンスは、そのソフトウェアとともに配布される他のソフトウェア に制約を加えてはなりません。たとえば、同じ媒体で配布される 他のソフトウェアがすべてフリーソフトウェアでなければならないと 要求してはいけません。
10.「 フリーなライセンスの例」…GPL、 BSD、および Artistic ライセンスは私たちがフリーと判断しているライセンスの例です。

図 1　The Debian Free Software Guidelines (DFSG)



5 Debian の OCaml 環境で開発する関数型言語インタプリタ

日比野 啓

5.1 OCaml はどんな言語

静的型の型推論言語というのが今の私の認識です。関数型言語らしい機能が注目されますが、それ以外のスタイルも良く利用されます。

ここではこの記事を読みすすめるにあたって必要そうな OCaml の機能について簡単に紹介します。

5.1.1 値

以下、対話環境の入出力を前提に話をすすめます。

ocaml-interp パッケージの /usr/bin/ocaml が対話環境のプログラムです。

`#` は対話環境のプロンプトです。ユーザーの入力の終わりを対話環境に認識してもらうために `;;` を入力します。なので対話環境の`#` プロンプトの後ろから`;;` までがユーザーの入力です。`-` :から始まる内容は対話環境の出力です。

```
# 1;;
- : int = 1
# "abc";;
- : string = "abc"
# (1, "abc");;
- : int * string = (1, "abc")
# (1, ("abc", 2), 3);;
- : int * (string * int) * int = (1, ("abc", 2), 3)
# [ "a"; "b"; "c"];;
- : string list = ["a"; "b"; "c"]
# "a" :: "b" :: "c" :: [];;
- : string list = ["a"; "b"; "c"]
```

値を表す式を入力すると対話環境は型の名前と値を出力しています。 `1` は `int` 型、`"abc"`は `string` 型となっています。対話環境に式を入力したので評価が行なわれ、その結果としての出力です。

`(1, ("abc", 2), 3)` のようにコンマで区切って括弧でくくった値はタプルです。値と値の組を表現できます。型の名前は `*`で連ねます。任意の型の値を組にできる強力な機能です。

`[ ]` でリストを表現することができます。また、`::`は lisp でいうところの cons です。リストの要素は全て同じ型である必要があります。

5.1.2 レコードとバリアント

次は値を表現する前にあらかじめ型の定義が必要であるような値です。

```
# type vec_2d = { x : int; y : int };;
type vec_2d = { x : int; y : int; }
# { x = 1; y = 2 };;
- : vec_2d = {x = 1; y = 2}
# type int_or_string_or_none = I of int | S of string | N;;
type int_or_string_or_none = I of int | S of string | N
# I 3;;
- : int_or_string_or_none = I 3
# S "hello";;
- : int_or_string_or_none = S "hello"
# N;;
- : int_or_string_or_none = N
```

一つ目の例はレコードです。C の構造体のようなもので、フィールド名とフィールドの型を定義に記述します。`vec_2d` というレコードの型を定義して、`{ x = 1; y = 2 }`という値を評価させました。

二つ目はバリアントあるいは代数データ型と呼ばれている種類の型です。`int_or_string_or_none` という型を定義していて、その値は I というタグの付いた int か S というタグの付いた string か N というタグ のどれかということです。このタグのことを**コンストラクタ**と呼びます。

バリアントは再帰的な定義も可能で、この機能で簡単に木構造を表現できます。

```
# type str_tree = Leaf of string | Tree of (str_tree * str_tree);;
type str_tree = Leaf of string | Tree of (str_tree * str_tree)
# Leaf "abc";;
- : str_tree = Leaf "abc"
# Tree (Leaf "a", Tree (Leaf "b", Leaf "c"));;
- : str_tree = Tree (Leaf "a", Tree (Leaf "b", Leaf "c"))
```

5.1.3 関数

次は関数を表現する値です。

```
# (fun x -> x + 1);;
- : int -> int = <fun>
# (fun x -> x);;
- : 'a -> 'a = <fun>
```

`(fun x -> x + 1)` は関数です。型が `int -> int` と出力されていますが、これは int を受けとって int を返す関数という意味です。`+` の引数は int と int なので結果として `x` は `int`、`fun x -> x + 1` は `int -> int` というように型の推論が行なわれています。

二番目の `(fun x -> x)` も関数です。型が`'a -> 'a` と出力されていますが、これは何かある型の値を受けとって、同じ型の値を返す関数という意味です。この `'a -> 'a` の関数は任意の型に対して適用が可能なので、全ての型についてこの関数が定義されている (`int -> int` も `string -> string` も ... etc) のと同様の効果があります。このような型を多相型と呼びます。

5.1.4 束縛

`let` を使って値を変数に束縛します。以下は `v` にたいして整数を `f` に対して関数を束縛しています。

```
# let v = 123;;
val v : int = 123
# v;;
- : int = 123
# let f x y = (x + y) * (x - y);;
val f : int -> int -> int = <fun>
# f 5 3;;
- : int = 16
```

5.1.5 パターン照合

OCaml は値の構造を分解しつつ変数を束縛することができます。次の例ではタプルを分解して束縛を行なっています。

```
# let (x, y) = (2, 3 + 4);;
val x : int = 2
val y : int = 7
# let (a, (b, c)) = (1, (2, 3));;
val a : int = 1
val b : int = 2
val c : int = 3
```

`match ... with` を使用すると、構造分解を試みつつ条件分岐することができます。

関数 `len` はリストが空かどうかを判定しながら再帰しています。束縛の定義時に自身の定義を使用するときには `let` ではなく `let rec` を使います。

関数 `what` は先程定義したバリアント `int_or_string_or_none` の値の種類によって返す文字列を変えています。

関数 `get_int` は `int_or_string_or_none` が int のときだけ Some int を返し、そうでないときは None を返します。'a option は組み込みの型で、ある型の値が有るかあるいは無いかを表現できるバリアントです。

```
# let rec len ls = match ls with [] -> 0 | x :: rest -> 1 + len rest;;
val len : 'a list -> int = <fun>
# len [1; 2; 3];;
- : int = 3
# let what v = match v with I _ -> "int" | S _ -> "string" | N -> "none";;
val what : int_or_string_or_none -> string = <fun>
# what (S "abc");;
- : string = "string"
# what N;;
- : string = "none"
# let get_int v = match v with I i -> Some i | _ -> None ;;
val get_int : int_or_string_or_none -> int option = <fun>
# get_int N;;
- : int option = None
# get_int (I 10);;
- : int option = Some 10
```

5.1.6 モジュール

プログラムの規模がおおきくなってくると名前空間が重要になってきます。 OCaml には module の機能があり名前空間を分けることができます。あるコンパイル単位が xyz.ml というファイル名だった場合、その中の定義は Xyz という module の中に配置されます。モジュール内にたとえば abc という名前があった場合、公開されていれば他のコンパイル単位のファイルからも Xyz.abc という名前でアクセスすることができます。

```
(* xyz.ml *)
let abc = "abc"
...


(* 他のファイル *)
... Xyz.abc ...
```

module の中にさらに module を定義することもできます。 xyz.ml の中に作った module SubXyz は、 Xyz.SubXyz という名前の module です。

定義済みのモジュールを使って module を定義することもできます。組み込みの module である List を使って Xyz.L を定義しています。

```
(* xyz.ml *)
module SubXyz =
  struct
  ...
  end

module L = List
```

5.2 関数型言語のインタプリタの簡単な作りかた

関数型言語とは何でしょうか。ここでは関数を値としてあつかえる言語ということにして、そのような言語のインタプリタを作る話をします。

5.2.1 環境を渡すナイーブなインタプリタ実装

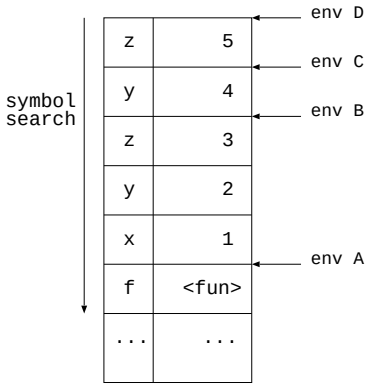
■let の環境

例えば以下のようなプログラムを考えます。

```
;; scheme function
(define (f)
  ;; env A
  (let ((x 1) (y 2) (z 3))
    ;; env B
    (let ((y 4))
      ;; env C
      (let ((z 5))
        ;; env D
        (+ x y z)))))
```

```
(* OCaml function *)
let f () =
  (* env A *)
  let (x, y, z) = (1, 2, 3) in
    (* env B *)
  let y = 4 in
    (* env C *)
  let z = 5 in
    (* env D *)
    x + y + z
```

変数の値の解決のためのテーブルを環境 (environment) と呼ぶことにして、次の図ような構造になっていると考えてみます。変数の検索は上から下に行なわれるとすると、それぞれのコメントの位置の環境は矢印の位置を参照していると考えてよいはずです。



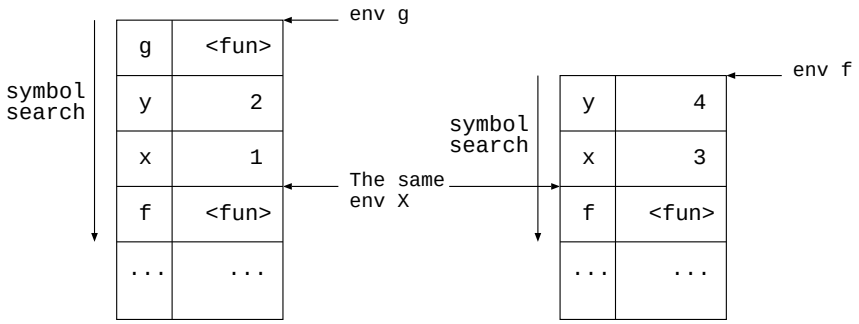
■関数呼び出しにおける環境

さらに以下のようなプログラムを考えます。

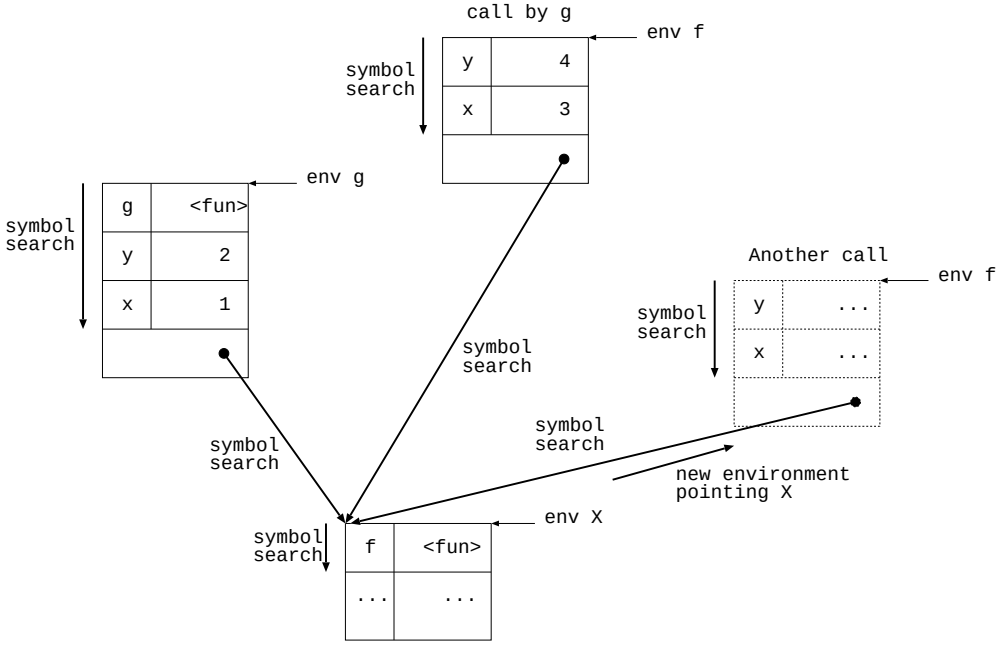
```
(define (f x y)
  ;; env f
  (* x y 2))

;;env X

(let ((x 1) (y 2))
  (define (g)
    ;; env g
    (f 3 4))
  (g))

;;; may be another call of (f x y)
```

```
let rec f x y =
  (* env f *)
  x * y * 2

(* env X *)

let (x, y) = (1, 2)
let rec g () =
  (* env g *)
  f 3 4

let v = g ()

(* may be another call of f x y *)
```

するとそれぞれのコメントの位置の環境は次の図のようになっているはずです。



ここで環境 X は同じ環境なので、共有させることにすると次のような構造を考えることができます。ここで注意する必要があるのは、環境 f は関数 f が呼び出される度に異なるということです。 f の定義位置である環境 X は共通ですが、環境 f は関数 f が呼び出される度に環境 X を指す環境を伸長する必要があります。



■クロージャの環境

以下のようなプログラムを考えます。

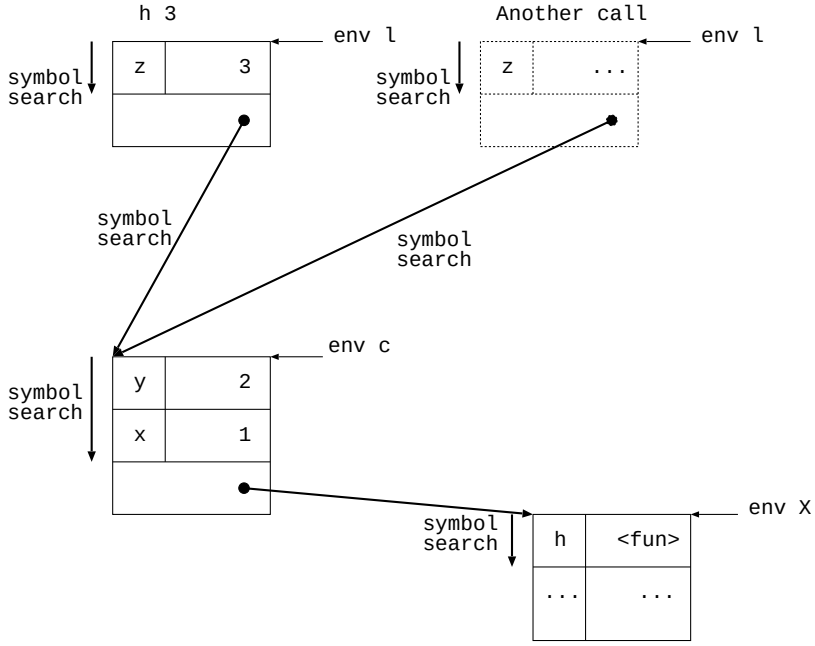
```
(define h
  (let ((x 1) (y 2))
    ;; env c
    (let ((f (lambda (z)
               ;; env l
               (+ x y z))))
      f)))

;; env X

(h 3)
;;; may be another call of (h x)
```

```
let rec h =
  let (x, y) = (1, 2) in
  (* env c *)
  let f z =
    (* env l *)
    x + y + z
  in f

(* env X *)

let v = h 3
(* may be another call of h x *)
```

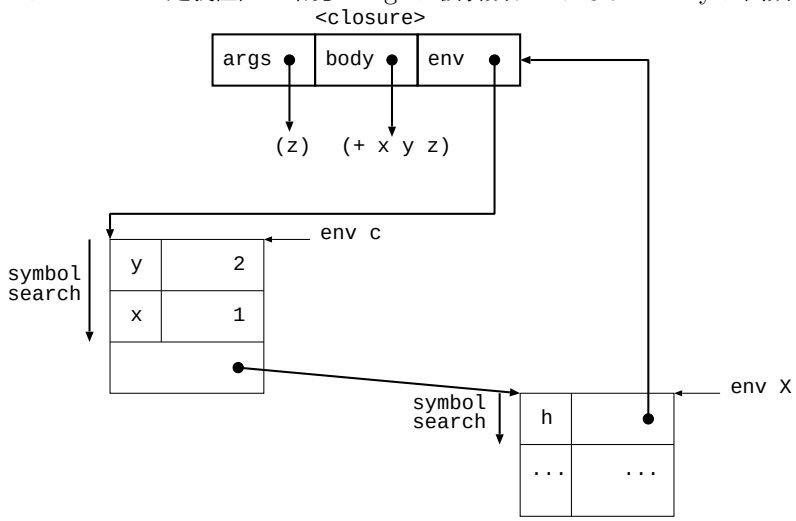
関数が変数 h に束縛されています。しかも h は x, y を束縛している let の外側にあるにもかかわらず、呼び出した際には x, y の値を評価します。

この let のような機能をレキシカルスコープと呼び、またこのような関数をレキシカルクロージャ (lexical closure) あるいは単にクロージャ (closure) と呼びます。

上の例での環境を考えてみると次のような構造になっているはずです。



h に束縛されている closure はあきらかに環境 c を知っている必要があります。なぜなら呼び出し時には環境 c を指す環境を生成しなければならないからです。したがって closure は以下のような構造になっていると考えることができます。env は closure の定義位置の環境で args は仮引数リストそして body は関数本体の式です。



式の評価に必要な環境を評価器内で渡しながら、 let や関数呼び出しの際には環境を伸長する方針を取ると、比較的簡単にインタプリタを実装することができます。また、環境を保持する構造を考えれば closure を実現することもできます。

参考資料: `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/class/isle4-05w/text/eopl003.html`

5.3 OCaml での lexing と parsing - ocamllex と ocamlyacc

5.3.1 ocamllex

ocamllex は OCaml に付属している字句解析関数生成器 (lexer generator) で、 OCaml から呼び出せる lexer を生成してくれます。以下のように C 言語でもおなじみの lex, flex と似たような使用感になっています。

```
{
  (* header *)
  (* Lexing のルール部分で参照したい内容を OCaml で書く *)
}

(* 文字列パターンの定義 *)

(* 字句解析 (lexing) のルール記述 *)

{
  (* trailer *)
  (* ルール部分で生成された関数を参照する内容を OCaml で書く *)
}
```

5.3.2 ocamlyacc

同様に、 ocamlyacc は OCaml に付属している構文解析関数生成器 (parser generator) で、 OCaml から呼び出せる parser を生成してくれます。こちらもやはり、以下のように yacc, bison と似たような使用感になっています。

```
%{
  (* header *)
  (* 構文木生成処理で参照したい内容を OCaml で書く *)
%}
  /* declarations */
  /* 終端記号の型や構文木の root の宣言 */
%%
  /* rules */
  /* 文脈自由文法と構文木生成処理を記述 */
%%
  (* trailer *)
  (* 生成された parser の関数を参照する内容を OCaml で書く *)
```

5.3.3 ocamllex と ocamlyacc の使用例

ocamllex と ocamlyacc を併用する場合には、 ocamlyacc で生成させた parser の終端記号の定義を lexer から参照させるようにするのがもっとも単純な使い方です。

以下が S 式 parser を生成させる例です。 sParser.mly を ocamlyacc で処理すると sParser.ml が生成され、sLexer.mll を ocamllex で処理すると sLexer.ml が生成されます。

■sParser.mly　型ごとに終端記号を %token で宣言し、 %start と %type で構文木の root とその型を指定します。 root の名前が構文解析関数の名前になります。

yacc と同じ要領で文脈自由文法を記述していきます。 expr は真偽値、数値、文字列であるか、あるいは、 expr ドット対または expr を並べたもの を括弧で括ったものという定義になっています。

```
%{
  (* sParser.mly *)

  module C = SCons

%}

/* File sparser.mly */
%token LPAREN RPAREN DOT_SYMBOL EOL BOOL_TRUE BOOL_FALSE
%token <SCons.s_int> INT
%token <float> FLOAT
%token <string> SYMBOL
%token <string> STRING
%start expr
%type <SCons.s_expr> expr
%%

expr_list:
  { C.Null }
| expr DOT_SYMBOL expr { C.Cons($1, $3) }
| expr expr_list { C.Cons($1, $2) }

expr:
| BOOL_TRUE               { C.Bool(true) }
| BOOL_FALSE              { C.Bool(false) }
| INT                     { C.Int($1) }
| FLOAT                   { C.Float($1) }
| SYMBOL                  { C.Symbol($1) }
| STRING                  { C.String($1) }
| LPAREN expr_list RPAREN { $2 }
```

■sLexer.mll　文字列パターンの定義では、正規表現の要領で文字集合や繰り返しの表現を利用して定義を作り、 let で名前を付けていきます。文字を ” でくくる以外は正規表現と同様です。定義した文字列パターンをさらに別の文字列パターンに再利用することができます。

ルール記述の部分では、 token の文字列パターンと token 生成式を lex の要領で記述していきます。キーワード rule の後の文字列が lexer の関数名になります。なのでここではその関数の名前は token です。

字句解析 (Lexing) の過程における入力ファイル内の位置は、 lexbuf の lex_start_p に記号 (token) の開始位置が、lex_curr_p に token の終了の次の位置が保持されています。 ocamllex デフォルトの動作では pos_cnum フィールドが更新されるのみなので、ファイル先頭からのバイト数しかわかりません。陽に行数の認識やタブによるカラム数の補正を行なう場合には、 lex_start_p と token をもとに lex_curr_p を修正してやる必要があります。[*9]

```
{
  (* sLexer.mll*)

  module LX = Lexing
  module P = SParser
  ... (* 中略 *)
  let fix_position lexbuf =
    let newline pos = {
      pos with
        LX.pos_lnum = pos.LX.pos_lnum + 1;
        LX.pos_cnum = pos.LX.pos_cnum + 1;
        LX.pos_bol = pos.LX.pos_cnum + 1;
    } in

    let tab pos = {
      pos with
        LX.pos_cnum = pos.LX.pos_cnum + 8 - (pos.LX.pos_cnum - pos.LX.pos_bol) mod 8
    } in

    let other pos = {
      pos with
        LX.pos_cnum = pos.LX.pos_cnum + 1
    } in

    let rec fix_pos_rec pos str =
      let len = (String.length str) in
        match (if len > 0 then (Some (str.[0]), String.sub str 1 (len - 1))
               else (None, "")) with
            (None, _) -> pos
          | (Some '\n', rest) -> fix_pos_rec (newline pos) rest
          | (Some '\t', rest) -> fix_pos_rec (tab pos) rest
          | (Some _, rest) -> fix_pos_rec (other pos) rest
    in
    let _ = lexbuf.LX.lex_curr_p <- fix_pos_rec (LX.lexeme_start_p lexbuf) (LX.lexeme lexbuf) in
      ()
}

/* 文字列パターンの定義 */
let str_esc = '\\'
let double_quote = '"'
let str_escaped_char = str_esc _
let str_char = [^ '\\' '"']
let str = double_quote (str_char | str_escaped_char)* double_quote

let left_paren = '('
let right_paren = ')'
let space = [' ' '\t' '\n' '\r']+
let dot_symbol = '.'
let bool_true  = '#' 't'
let bool_false = '#' 'f'

let int = '-'? ['0' - '9']+
let float = '-'? ['0' - '9']+ '.' ['0' - '9']* | '-'? ['0' - '9']* '.' ['0' - '9']+
let symbol = [^ '"' '(' ')' ' ' '\t' '\n' '\r']+

/* lexing のルール記述 */
rule token = parse
  | left_paren      { P.LPAREN }
  | right_paren     { P.RPAREN }
  | space      { fix_position lexbuf; token lexbuf }
  | dot_symbol      { P.DOT_SYMBOL }
  | bool_true       { P.BOOL_TRUE }
  | bool_false      { P.BOOL_FALSE }
  | int      { expr_integer (LX.lexeme lexbuf) }
  | float    { P.FLOAT(Pervasives.float_of_string(LX.lexeme lexbuf)) }
  | symbol   { P.SYMBOL(LX.lexeme lexbuf) }
  | str      { fix_position lexbuf; P.STRING(expr_string(LX.lexeme lexbuf)) }
  | eof      { raise Eof }
```

[*9] 次の lex_start_p は現在の lex_curr_p から引き継がれるので、 lex_curr_p を修正すれば十分です。

5.4 Haskell の Lexing

Haskell にはブロックの開始や終了の token や式の区切りの token を省略することができる layout rule という機能があります。そのため省略された token を lexing の過程で補ってやる必要があります。

まず、通常と同様に lexing を行なって token 列を生成し、その token 列に規則に従って token を補うというように、2 段階の工程を行ないます。

5.4.1 layout なしの Lexing

■lexer0.mll header 部分

まずは.mll の header 部分です。

後から layout rule において必要となるカラム数を数えあげる処理ために位置情報の修正を行なう関数 (fix_position) を定義しています。また、 Haskell の文字および文字列リテラルはリテラル内のルールが複雑度の高い仕様なので、別の lexer(後述) を呼び出しつつ実際の文字列表現を構成する関数 (decode_char, decode_string) を準備しています。

■fix_position 位置情報の修正

```
{
  (* lexer0.mll header 部分 *)
  module LX = Lexing
  module P = Parser
  ... (* 中略 *)
  let fix_position lexbuf =
    let newline pos =
      { pos with
          LX.pos_lnum = pos.LX.pos_lnum + 1;
          LX.pos_cnum = pos.LX.pos_cnum + 1;
          LX.pos_bol = pos.LX.pos_cnum + 1;
      } in

    let tab pos =
      { pos with
          LX.pos_cnum = pos.LX.pos_cnum + 8 - (pos.LX.pos_cnum - pos.LX.pos_bol) mod 8
      } in

    let other pos =
      { pos with
          LX.pos_cnum = pos.LX.pos_cnum + 1
      } in

    let rec fix_pos_rec pos str =
      let len = (String.length str) in
        match (if len > 0 then (Some (str.[0]), String.sub str 1 (len - 1))
               else (None, "")) with
            (None, _) -> pos
          | (Some '\n', rest) -> fix_pos_rec (newline pos) rest
          | (Some '\t', rest) -> fix_pos_rec (tab pos) rest
          | (Some _, rest) -> fix_pos_rec (other pos) rest
    in
    let _ = lexbuf.LX.lex_curr_p <- fix_pos_rec (LX.lexeme_start_p lexbuf) (LX.lexeme lexbuf) in
      ()
  ... (* 中略 *)
```

■decode_char, decode_string **文字および文字列デコーダー**

```
  ... (* 中略 *)
  let decode_cexpr cexpr =
    let fchar = String.get cexpr 0 in
    let escexp = String.sub cexpr 1 ((String.length cexpr) - 1) in
    let fmatch exp str = Str.string_match (Str.regexp exp) str 0 in
      if fchar = '\\' then
        match escexp with
            "NUL"   -> Some '\x00'
          | "SOH" | "^A"   -> Some '\x01'
          | "STX" | "^B"   -> Some '\x02'
        ... (* 中略 *)
          | "RS"  | "^^"   -> Some '\x1e'
          | "US"  | "^_"   -> Some '\x1f'
          | "SP"           -> Some ' '

          | "\\"           -> Some '\\'
          | "\""           -> Some '"'
          | "'"            -> Some '\''

          | "DEL"          -> Some '\x7f'

          | _ when fmatch "^[0-9]+$" escexp
              -> Some (Char.chr (int_of_string escexp))
          | _ when fmatch "^[xX][0-9a-zA-Z]+$" escexp
              -> Some (Char.chr (int_of_string ("0" ^ escexp)))
          | _ when fmatch "^[oO][0-7]+$" escexp
              -> Some (Char.chr (int_of_string ("0" ^ escexp)))

          | _ -> None

      else Some fchar

  let decode_char lexbuf =
    let cstr = LX.lexeme lexbuf in
    let len = String.length cstr in
      match decode_cexpr (String.sub cstr 1 (len - 2)) with
          Some c -> c
        | None   -> failwith (F.sprintf "Unkown char expression %s" cstr)

  let decode_string lexbuf =
    let sexpr = LX.lexeme lexbuf in
    let len = String.length sexpr in
    let strlbuf = Lexing.from_string (String.sub sexpr 1 (len - 2)) in
    let rec decode result =
      match HsStr.char strlbuf with
          HsStr.Eos -> result
        | HsStr.Char cstr ->
            if cstr = "\\&" then decode (result ^ "&")
            else decode (result ^
                           match (decode_cexpr cstr) with
                               None -> failwith (F.sprintf "Unkown char expression '%s' in literal string" cstr)
                             | Some c -> (String.make 1 c))
        | HsStr.Gap g -> decode result
    in decode ""
}
```

■lexer0.mll **文字列パターン定義部分**

次に文字列パターンの定義です。

行数はちょっと多いですが、特に難しいところはありません。問題のリテラル文字列ですが、リテラル文字列部分の文字列パターン自体は問題なく表現できています。しかし、リテラルで表現される文字列自体を復元するのが複雑なので前記および後述のような準備が必要になります。

```
/* lexer0.mll 文字列パターン定義部分 */
let special = ['(' ')' ',' ';' '[' ']' '`' '{' '}']

let space = ' '
let newline = ("\r\n"|['\n' '\r'])
let tab = '\t'

let dashes = '-' '-' '-'*

let ascSmall = ['a'-'z']
let small = ascSmall | '_'
let ascLarge = ['A'-'Z']
let large = ascLarge

let plus = '+'
let minus = '-'
let exclamation = '!'
let ascSymbol_nbs = [ '!' '#' '$' '%' '&' '*' '+' '.' '/' '<' '=' '>' '?' '@' '^' '|' '-' '~' ]
let ascSymbol = ascSymbol_nbs | '\\'
let symbol = ascSymbol

let ascDigit = ['0'-'9']
let digit = ascDigit

let octit = ['0'-'7']
let hexit = ascDigit | ['a'-'z' 'A'-'Z']

let decimal = (digit)+
let octal = (octit)+
let hexadecimal = (hexit)+

let exponent = ['e' 'E'] ['+' '-']? decimal
let float = decimal '.' decimal exponent? | decimal exponent

let graphic = small | large | symbol | digit | special | [':' '"' '\'']
let any = graphic | space | tab

let comment = dashes ((space | tab | small | large | symbol | digit | special | [':' '"' '\'']) (any)*)? newline

let whitechar = newline | space | tab
let whitestuff = whitechar | comment
let whitespace = (whitestuff)+

(*
let lwhitechar = space | tab
let lwhitestuff = lwhitechar | comment
let lwhitespace = (lwhitestuff)+
*)

let char_gr = small | large | ascSymbol_nbs | digit | special | [':' '"']
let str_gr  = small | large | ascSymbol_nbs | digit | special | [':' '\'']

let charesc = ['a' 'b' 'f' 'n' 'r' 't' 'v' '\\' '"' '\'']
let str_charesc = charesc | '&'
let cntrl = ascLarge | ['@' '[' '\\' ']' '^' '_']
let gap = '\\' (whitechar)+ '\\'
(* let gap = '\\' (lwhitechar | newline)+ '\\' *)

let ascii = ('^' cntrl) | "NUL" | "SOH" | "STX" | "ETX" | "EOT" | "ENQ" | "ACK"
  | "BEL" | "BS" | "HT" | "LF" | "VT" | "FF" | "CR" | "SO" | "SI" | "DLE"
  | "DC1" | "DC2" | "DC3" | "DC4" | "NAK" | "SYN" | "ETB" | "CAN"
  | "EM" | "SUB" | "ESC" | "FS" | "GS" | "RS" | "US" | "SP" | "DEL"

let escape = '\\' ( charesc | ascii | decimal | 'o' octal | 'x' hexadecimal )
let str_escape = '\\' ( str_charesc | ascii | decimal | 'o' octal | 'x' hexadecimal )

let char = '\'' (char_gr | space | escape) '\''
let string = '"' (str_gr | space | str_escape | gap)* '"'

let varid = small (small | large | digit | '\'')*
let conid = large (small | large | digit | '\'')*

let varsym = symbol (symbol | ':')*
let consym = ':' (symbol | ':')*

let modid = conid
```

■lexer0.mll **ルール記述部分**

最後にルール記述です。

スペース、タブ、改行などを含んでいる whitespace や string のところで fix_position を呼んで位置情報を補正しています。また char や string のリテラルから文字や文字列を構成するために decode_char, decode_string を呼び出しています。

```
(* lexer0.mll ルール記述部分 *)
rule token = parse
  | '('  { P.SP_LEFT_PAREN(loc lexbuf) }
  | ')'  { P.SP_RIGHT_PAREN(loc lexbuf) }
  | ','  { P.SP_COMMA(loc lexbuf) }
  | ';'  { P.SP_SEMI(loc lexbuf) }
  | '['  { P.SP_LEFT_BRACKET(loc lexbuf) }
  | ']'  { P.SP_RIGHT_BRACKET(loc lexbuf) }
  | '`'  { P.SP_B_QUOTE(loc lexbuf) }
  | '{'  { P.SP_LEFT_BRACE(loc lexbuf) }
  | '}'  { P.SP_RIGHT_BRACE(loc lexbuf) }
     (** special tokens *)

  | "case"     { P.K_CASE(loc lexbuf) }
  | "class"    { P.K_CLASS(loc lexbuf) }
  | "data"     { P.K_DATA(loc lexbuf) }
  | "default"  { P.K_DEFAULT(loc lexbuf) }
  | "deriving" { P.K_DERIVING(loc lexbuf) }
  | "do"       { P.K_DO(loc lexbuf) }
  | "else"     { P.K_ELSE(loc lexbuf) }
  | "if"       { P.K_IF(loc lexbuf) }
  | "import"   { P.K_IMPORT(loc lexbuf) }
  | "in"       { P.K_IN(loc lexbuf) }
  | "infix"    { P.K_INFIX(loc lexbuf) }
  | "infixl"   { P.K_INFIXL(loc lexbuf) }
  | "infixr"   { P.K_INFIXR(loc lexbuf) }
  | "instance" { P.K_INSTANCE(loc lexbuf) }
  | "let"      { P.K_LET(loc lexbuf) }
  | "module"   { P.K_MODULE(loc lexbuf) }
  | "newtype"  { P.K_NEWTYPE(loc lexbuf) }
  | "of"       { P.K_OF(loc lexbuf) }
  | "then"     { P.K_THEN(loc lexbuf) }
  | "type"     { P.K_TYPE(loc lexbuf) }
  | "where"    { P.K_WHERE(loc lexbuf) }
  | "_"        { P.K_WILDCARD(loc lexbuf) }
     (** reservedid *)

  | ".."       { P.KS_DOTDOT(loc lexbuf) }
  | ":"        { P.KS_COLON(loc lexbuf) }
  | "::"       { P.KS_2_COLON(loc lexbuf) }
  | "="        { P.KS_EQ(loc lexbuf) }
  | "\\"       { P.KS_B_SLASH(loc lexbuf) }
  | "|"        { P.KS_BAR(loc lexbuf) }
  | "<-"       { P.KS_L_ARROW(loc lexbuf) }
  | "->"       { P.KS_R_ARROW(loc lexbuf) }
  | "@"        { P.KS_AT(loc lexbuf) }
  | "~"        { P.KS_TILDE(loc lexbuf) }
  | "=>"       { P.KS_R_W_ARROW(loc lexbuf) }
     (** reservedop *)

  | "as"              { P.K_AS(loc lexbuf) }  (** maybe varid *)
  | "qualified"       { P.K_QUALIFIED(loc lexbuf) }  (** maybe varid *)
  | "hiding"          { P.K_HIDING(loc lexbuf) }  (** maybe varid *)
  | varid      { P.T_VARID(LX.lexeme lexbuf, loc lexbuf) }
  | conid      { P.T_CONID(LX.lexeme lexbuf, loc lexbuf) }
     (** identifiers or may be qualified ones *)

  | whitespace  { fix_position lexbuf; P.WS_WHITE(loc lexbuf) }  (** comment begining with dashes is not varsym *)
     (** white spaces *)

  | plus       { P.KS_PLUS(loc lexbuf) }  (** maybe varsym *)
  | minus      { P.KS_MINUS(loc lexbuf) } (** maybe varsym *)
  | exclamation  { P.KS_EXCLAM(loc lexbuf) } (** maybe varsym *)
  | varsym     { P.T_VARSYM(LX.lexeme lexbuf, loc lexbuf) }
  | consym     { P.T_CONSYM(LX.lexeme lexbuf, loc lexbuf) }
     (** symbols or may be qualified ones *)

  | modid '.' varid   { P.T_MOD_VARID(decode_with_mod lexbuf, loc lexbuf) }
  | modid '.' conid   { P.T_MOD_CONID(decode_with_mod lexbuf, loc lexbuf) }
  | modid '.' varsym  { P.T_MOD_VARSYM(decode_with_mod lexbuf, loc lexbuf) }
  | modid '.' consym  { P.T_MOD_CONSYM(decode_with_mod lexbuf, loc lexbuf) }
     (** qualified xx *)

  | char      { P.L_CHAR(decode_char lexbuf, loc lexbuf) }
  | string    { fix_position lexbuf; P.L_STRING(decode_string lexbuf, loc lexbuf) }

  | decimal | ('0' ['o' 'O'] octal) | ('0' ['x' 'X'] hexadecimal)
       { P.L_INTEGER(Int64.of_string(LX.lexeme lexbuf), loc lexbuf) }

  | float      { P.L_FLOAT(float_of_string(LX.lexeme lexbuf), loc lexbuf) }

  | eof        { P.EOF(loc lexbuf) }
  ... /* 以下略 */
```

■hsStr.mll

文字列の lexer です。

ここでの token は文字列リテラル内の 1 文字の表現あるいはギャップ (gap) です。 Haskell では 1 つの文字列リテラルを中断して、間に空白や改行やコメントを記述した後に、再開することができます。この空白や改行やコメントの部分が gap です。

ここで定義された char 関数を利用して decode_string 関数は文字列を構成していくようになっています。

```
{
  (* hsStr.mll *)
  module LX = Lexing

  type ct =
      Char of string
    | Gap of string
    | Eos
}

let special = ['(' ')' ',' ';' '[' ']' '`' '{' '}']

let space = ' '
let newline = ("\r\n"|['\n' '\r'])
let tab = '\t'

let ascSmall = ['a'-'z']
let small = ascSmall
let ascLarge = ['A'-'Z']
let large = ascLarge

let ascSymbol_nbs = [ '!' '#' '$' '%' '&' '*' '+' '.' '/' '<' '=' '>' '?' '@' '^' '|' '-' '~' ]

let ascDigit = ['0'-'9']
let digit = ascDigit

let octit = ['0'-'7']
let hexit = ascDigit | ['a'-'z' 'A'-'Z']

let decimal = (digit)+
let octal = (octit)+
let hexadecimal = (hexit)+

let lwhitechar = space | tab

let str_gr  = small | large | ascSymbol_nbs | digit | special | [':' '\'']

let charesc = ['a' 'b' 'f' 'n' 'r' 't' 'v' '\\' '"' '\'']
let str_charesc = charesc | '&'
let cntrl = ascLarge | ['@' '[' '\\' ']' '^' '_']
let gap = '\\' (lwhitechar | newline)+ '\\'

let ascii = ('^' cntrl) | "NUL" | "SOH" | "STX" | "ETX" | "EOT" | "ENQ" | "ACK"
  | "BEL" | "BS" | "HT" | "LF" | "VT" | "FF" | "CR" | "SO" | "SI" | "DLE"
  | "DC1" | "DC2" | "DC3" | "DC4" | "NAK" | "SYN" | "ETB" | "CAN"
  | "EM" | "SUB" | "ESC" | "FS" | "GS" | "RS" | "US" | "SP" | "DEL"

let str_escape = '\\' ( str_charesc | ascii | decimal | 'o' octal | 'x' hexadecimal )

rule char = parse
  | str_gr | space | str_escape  { Char(LX.lexeme lexbuf) }
  | gap                          { Gap(LX.lexeme lexbuf) }
  | eof                          { Eos }
```

参考資料: `http://www.sampou.org/haskell/report-revised-j/lexemes.html`

5.4.2 Haskell の layout rule

この節の始めにも書いたように layout rule は、 token 列にさらに token を補ってやる処理です。まずは補うルールを確認してみましょう。以下に、 Haskell 98 Language Report の改訂版の和訳[*10]から引用してみます。

..... 引用ここから .....

レイアウトの影響は、この節では、レイアウトを用いているプログラムに、どのようにして、ブレースとセミコロンを追加するかを記述することによって指定する。この仕様は、変換を行う関数 L の形をとる。 L への入力は

- この Haskell レポートの字句構文で指定されたような字句の並びで、以下のような追加トークンがついているもの。
  - キーワード let、 where、 do あるいは of のあとに字句 { が続かない場合、トークン $\{n\}$ をキーワードの後に挿入する。ここで n は、もし次の字句があればそれのインデント、または、ファイルの終端に到達していれ

[*10] http://www.sampou.org/haskell/report-revised-j/syntax-iso.html#layout

ば 0 である。

– モジュールの最初の字句が { あるいは module では ないとき、その字句の前に $\{n\}$ を置く。ここで、 n はその字句のインデントである。

– 同一行で、字句の開始の前には白空白しかないとき、この字句の前に $< n >$ を置く。ここで n はこの字句のインデントで、 前の二つの規則の結果、その前には $\{n\}$ が置かれていない。 (注意: 文字列リテラルは複数行にまたがることがある – 2.6 節。したがって、

```
f = ("Hello \
       \Bill", "Jake")
```

では、\Bill の前に $< n >$ は挿入されることはない。なぜなら、完全な字句の開始場所ではないからだ。また、, の前にも $< n >$ は置かれることはない。なぜなら、その前に 白空白以外のものがあるからだ。)

- 「 レイアウト文脈」のスタックのそれぞれの要素は以下のどれかである。
  – ゼロ、これは文脈を明示的に囲うこと (たとえばプログラマが開ブレース を用意した場合) を示す。もし最も内側の文脈が 0 なら、囲まれた文脈が 終了するか、新しい文脈がプッシュされるまで、レイアウトトークンは挿入されない。
  – 正の整数、これは囲まれたレイアウト文脈のインデントカラム数

字句の「 インデント」は字句の最初の文字のカラム数である。ひとつの行のインデントとは最も左にある字句のインデントを表す。このカラム数を決定するために以下のような規約をもつ固定幅のフォントを仮定する。

- 改行、リターン、ラインフィードおよびフォームフィード文字はすべて新しい行を開始する
- 最初のコラムは 0 ではなく 1 である
- タブストップは 8 文字文ずつの位置にある
- タブ文字は現在位置から次のタブストップ位置までそろえるのに必要なだけの空白を挿入する。

レイアウトルールにあわせるために、ソースプログラム中の Unicode 文字は ASCII 文字と同じ幅の固定幅であると看倣す。しかしながら、見た目との混 乱を避けるためプログラマは暗黙のレイアウトの意味が非空白文字の幅に依 存するようなプログラムを書かないようにすべきである。

■適用

L tokens [ ] は、 tokens のレイアウトに関知しない変換をもたらす。ここで、 tokens はモジュールの字句解析および上述のようにカラム数表示子を追加した結果である。 L の定義は以下のとおり、ここでは「 :」をストリーム構築操作子として使い、「 [ ]」は空のストリームである。

```
L (<n>:ts) (m:ms)   = ; : (L ts (m:ms))  if m = n
                    = } : (L (<n>:ts) ms) if n < m
L (<n>:ts) ms       = L ts ms
L ({n}:ts) (m:ms)   = { : (L ts (n:m:ms)) if n > m   (Note 1)
L ({n}:ts) []       = { : (L ts [n]) if n > 0        (Note 1)
L ({n}:ts) ms       = { : } : (L (<n>:ts) ms)        (Note 2)
L (}:ts) (0:ms)     = } : (L ts ms)                  (Note 3)
L (}:ts) ms         = parse-error                    (Note 3)
L ({:ts) ms         = { : (L ts (0:ms))              (Note 4)
L (t:ts) (m:ms)     = } : (L (t:ts) ms) if m /= 0 and parse-error(t)  (Note 5)
L (t:ts) ms         = t : (L ts ms)
L [] []             = []
L [] (m:ms)         = } : L [] ms if m /=0           (Note 6)
```

.....引用ここまで 以下略 .....

だいぶ長くなってしまったので Note は省略です。

わかりにくいですが、ここでも内部的には 2 段階になっています。

まず元の token 列に一つ目の操作を適用します。以下のような規則だと考えるとわかりやすいかもしれません。

- let, where, do, of の後に { が無い場合には代わりにブロックの開始をあらわす token $\{n\}$ を挿入。
- ファイルの先頭もモジュール宣言が省略されていて { が無い場合には token $\{n\}$ でブロック開始。
- インデントのレベルで後からブロックを認識するために token $< n >$ を挿入しておく。

つぎに二つ目の操作である関数 L を適用します。

L はもとの token 列とインデントレベルのスタックを引数にとり、もとの token 列に token を挿入したものを返す関数です。やはり以下のような規則だと考えるとわかりやすいかもしれません。

- インデントレベル $< n >$ が同じレベルのブロック内であれば (スタック参照) ; を挿入して式を終了、ブロックを継続
- インデントレベル $< n >$ の方が浅ければ } を挿入してブロックを終了
- インデントレベル $< n >$ があって上のどちらでもなければ式を継続
- あるブロック内で (スタック参照) よりインデントレベルの深いブロック開始 $\{n\}$ があったら { を挿入しブロックを開始。ブロック開始をあらわすインデントレベル n をスタックに積む
- ブロック開始 $\{n\}$ が最も外側でも同様にブロック開始。 { を挿入し、 n をスタックに積む
- ブロック開始 $\{n\}$ があって上のどちらでもなければ { および } を挿入し空のブロックを作る。実はブロック開始ではなかったということがここでわかるので代わりに $< n >$ を挿入する。
- } があったらスタックから 0 を取り出す
- } があって 0 を降ろせなかったら parse error
- { があったらスタックに 0 を積む
- 上のどれでもなく、またブロック内であり、ブロックを継続すると parse error になるときは } を挿入してブロックを閉じる。
- 上のどれでもなく、またブロック内であるときは parse error にならない限りブロックを継続。
- トークンもスタックも空ならおわり
- トークンが空でスタックに 0 でない値が残っているなら } を挿入してスタックから取り出す。(0 があったらエラー)

参考資料: `http://www.sampou.org/haskell/report-revised-j/syntax-iso.html#layout`

■OCaml での実装

「 parse error にならない限りブロックを継続」のルールはかなり実装がやっかいでした。今回利用した ocamlyacc は yacc や bison と同じように LALR(1) の parser generator となっており、基本的には token 列を途中まで読み込んだ位置とその token および一つ先の token によって構文解析中の次の動作を決定しています。ここで出てきたような parse error になるまでブロックが閉じるかわからないような仕様とはあまり相性が良くありません。 backtrack を行なえるような parser ならこのような仕様に対してもより簡単に対応できると考えられます。今回は token 列の token ごとに parse error のフラグを付加してやりなおしを行なうことで対応しました。実行効率はよくありませんが問題になるほど token 数が多くならなければ大丈夫そうです。将来的には Packrat parsing のような方法で parser を書き直してみたいところです。

OCaml で上 layout rule を実装した関数が次のような感じです。 token 列への一つ目の操作と二つ目の操作を順番に載せています。 P.BLK_OPEN が $\{n\}$ で P.BLK_LEVEL が $< n >$ にあたるものです。もとの定義と似たよう記述で表現できているのがわかるでしょうか。

```
let all_token_rev_list lexbuf =
  let unget_s = S.create () in
  let get_token () = L0.token lexbuf in
  let blk_level_pair tk =
    let loc = L0.get_location tk in (loc.T.start_p.T.col + 1, loc) in
  let eof_token_p = (function P.EOF(_) -> true | _ -> false) in

  let rec scan_start () =
    match get_token () with
        (P.SP_LEFT_BRACE _ | P.K_MODULE _) as start -> start
      | P.WS_WHITE _ -> scan_start ()
      | other ->
          let _ = S.push other unget_s in
            P.BLK_OPEN (blk_level_pair other)
  in

  let scan_next prev =
    let rec scan_next_rec () =
      let cur =
        if (S.is_empty unget_s) then (get_token ())
        else (S.pop unget_s) in

        match (prev, cur) with
            (_, (P.EOF(_) as eoft)) -> eoft
          | (_, P.WS_WHITE(_)) -> (scan_next_rec ())
          | ((P.K_LET(_) | P.K_WHERE(_) | P.K_DO(_) | P.K_OF(_)), (P.SP_LEFT_BRACE(_) as lbr)) -> lbr
          | ((P.K_LET(_) | P.K_WHERE(_) | P.K_DO(_) | P.K_OF(_)), tk) ->
              let (_, (level, loc)) = (S.push tk unget_s, blk_level_pair tk) in
                P.BLK_OPEN((if (eof_token_p tk) then 0 else level), loc)
          | (_, tk) ->
              let (_, loc) as p = blk_level_pair tk in
                if (loc.T.start_p.T.line
                    - (L0.get_location prev).T.end_p.T.line) > 0 then
                  let _ = S.push tk unget_s in P.BLK_LEVEL p
                else tk
    in (scan_next_rec ())
  in
    (LST.fold_left
       (fun r a -> ((a, new_err_flag ()) :: r))
       []
       (LST.create_stream (scan_start ()) scan_next eof_token_p))
```

```
let rec layout istream levels =
  let push_new_token tok lform =
    LST.Cons ((tok, new_err_flag ()), lform)
  in

  let (tok, err) =
    match LST.peek istream with
        None -> raise Parsing.Parse_error
      | Some x -> x
  in
    match (tok, levels) with
        ((P.BLK_LEVEL (n, loc)), (m :: mstl as ms)) when m = n ->
          let addtk = P.SP_SEMI(loc) in
            push_new_token addtk (lazy (layout (LST.tl istream) ms))
      | ((P.BLK_LEVEL (n, loc)), m :: ms) when n < m  ->
          push_new_token (P.SP_RIGHT_BRACE(loc)) (lazy (layout istream ms))
      | ((P.BLK_LEVEL (n, _)), ms)                    -> layout (LST.tl istream) ms
      | ((P.BLK_OPEN (n, loc)), (m :: ms as levels)) when n > m  ->
          push_new_token (P.SP_LEFT_BRACE(loc)) (lazy (layout (LST.tl istream) (n :: levels))) (* Note 1 *)
      | ((P.BLK_OPEN (n, loc)), []) when n > 0        ->
          push_new_token (P.SP_LEFT_BRACE(loc)) (lazy (layout (LST.tl istream) [n])) (* Note 1 *)
      | ((P.BLK_OPEN (n, loc)), ms)                   ->
          push_new_token
            (P.SP_LEFT_BRACE(loc))
            (lazy (push_new_token
                     (P.SP_RIGHT_BRACE(loc))
                     (lazy (layout (push_new_token
                                      (P.BLK_LEVEL(n, loc))
                                      (lazy (LST.tl istream))) ms)))) (* Note 2 *)
      | ((P.SP_RIGHT_BRACE _ as rbr), 0 :: ms)        ->
          LST.Cons ((rbr, err), lazy (layout (LST.tl istream) ms)) (* Note 3 *)
      | ((P.SP_RIGHT_BRACE _), ms)                    -> raise Parsing.Parse_error (* Note 3 *)
      | ((P.SP_LEFT_BRACE _ as lbr), ms)              ->
          LST.Cons ((lbr, err), lazy (layout (LST.tl istream) (0 :: ms))) (* Note 4 *)

      | ((P.EOF loc as eoft), [])                     -> LST.Cons ((eoft, err), lazy (LST.Nil))
      | ((P.EOF loc), m :: ms) when m <> 0            ->
          push_new_token (P.SP_RIGHT_BRACE(loc)) (lazy (layout istream ms)) (* Note 6 *)

      | (t, (m :: mstl)) when m <> 0 && (!err)        ->
          err := false;
          push_new_token (P.SP_RIGHT_BRACE(L0.get_location t)) (lazy (layout istream mstl))
          (* parse-error(t) Note 5 case *)
      | (t, ((m :: mstl) as ms))                    ->
          LST.Cons ((t, err),
                    lazy (layout (LST.tl istream) ms))
      | (t, ms)                                       ->
          LST.Cons ((t, err),
                    lazy (layout (LST.tl istream) ms))
```

5.5 Haskell の Parsing

文脈自由文法の定義を全部載せてしまうと長すぎて大変なので、ここでは単項の expression の部分と、二項演算および二項演算パターンの定義を見ていくことにします。

5.5.1 Haskell の Expression

■単項の expression

単項の表現 aexp としてここで定義されているのは、変数、コンストラクタ、リテラル、 括弧でくくられた expression、 タプル、 リスト、 リスト内包表記、 括弧でくくられた left secion、括弧でくくられた right secion (section は二項演算式でどちらか充足しているもの)、 レコードの生成、 レコードをコピーして更新です。単項の表現は二項演算の引数、関数、関数の引数となり得ます。

```
/* parser.mly 単項 expression */
/*
 aexp    ->      qvar    (variable)
        |       gcon    (general constructor)
        |       literal
        |       ( exp )         (parenthesized expression)
        |       ( exp1 , ... , expk )   (tuple, k>=2)
        |       [ exp1 , ... , expk ]   (list, k>=1)
        |       [ exp1 [, exp2] .. [exp3] ]     (arithmetic sequence)
        |       [ exp | qual1 , ... , qualn ]   (list comprehension, n>=1)
        |       ( expi+1 qop(a,i) )      (left section)
        |       ( lexpi qop(l,i) )       (left section)
        |       ( qop(a,i)<-> expi+1 )   (right section)
        |       ( qop(r,i)<-> rexpi )    (right section)
        |       qcon { fbind1 , ... , fbindn }  (labeled construction, n>=0)
        |       aexp<qcon> { fbind1 , ... , fbindn }    (labeled update, n >= 1)
*/

aexp:
  qvar  { E.VarE $1 }   /*(variable)*/
| gcon  { E.ConsE $1 }  /*(general constructor)*/
| literal  { E.LiteralE $1 }
| SP_LEFT_PAREN exp SP_RIGHT_PAREN  { E.ParenE $2 }     /*(parenthesized expression)*/
| SP_LEFT_PAREN exp SP_COMMA exp_list SP_RIGHT_PAREN  { E.TupleE ($2 :: $4) }   /*(tuple, k>=2)*/
| SP_LEFT_BRACKET exp_list SP_RIGHT_BRACKET  { E.ListE ($2) }   /*(list, k>=1)*/
| SP_LEFT_BRACKET exp KS_DOTDOT SP_RIGHT_BRACKET  { E.ASeqE($2, None, None) }   /*(arithmetic sequence)*/
| SP_LEFT_BRACKET exp SP_COMMA exp KS_DOTDOT SP_RIGHT_BRACKET  { E.ASeqE($2, Some $4, None) }   /*(arithmetic sequence)*/
| SP_LEFT_BRACKET exp KS_DOTDOT exp SP_RIGHT_BRACKET  { E.ASeqE($2, None, Some $4) }    /*(arithmetic sequence)*/
| SP_LEFT_BRACKET exp SP_COMMA exp KS_DOTDOT exp SP_RIGHT_BRACKET  { E.ASeqE($2, Some $4, Some $6) }    /*(arithmetic sequence)*/
| SP_LEFT_BRACKET exp KS_BAR qual_list SP_RIGHT_BRACKET  { E.LCompE ($2, $4) }  /*(list comprehension, n>=1)*/

| SP_LEFT_PAREN op2_left_section SP_RIGHT_PAREN  { E.MayLeftSecE ($2) }         /*(left section)*/
| SP_LEFT_PAREN op2_right_section SP_RIGHT_PAREN  { E.MayRightSecE ($2) }       /*(right section)*/

| qcon SP_LEFT_BRACE fbind_list SP_RIGHT_BRACE  { E.LabelConsE ($1, OH.of_list $3) }    /*(labeled construction, n>=1)*/
| qcon SP_LEFT_BRACE SP_RIGHT_BRACE  { E.LabelConsE ($1, OH.create 0) }         /*(labeled construction, n=0)*/
| aexp SP_LEFT_BRACE fbind_list SP_RIGHT_BRACE  { E.LabelUpdE ($1, OH.of_list $3) }     /*(labeled update, n >= 1)*/
;

exp_list:
  exp SP_COMMA exp_list  { $1 :: $3 }
| exp  { [$1] }
;

qual_list:
  qual SP_COMMA qual_list  { $1 :: $3 }
| qual  { [$1] }
;

fbind_list:
  fbind SP_COMMA fbind_list  { $1 :: $3 }
| fbind  { [$1] }
;

qual:
  pat KS_L_ARROW exp  { LC.Gen($1, $3) }        /*(generator)*/
| K_LET decl_list  { LC.Let $2 }        /*(local declaration)*/
| exp  { LC.Guard $1 }  /*(guard)*/
;
```

5.5.2 二項演算子の優先順位

■二項演算 expression **および二項演算パターンの構文解析**

Haskell の二項演算子には level 0 から level 9 までの優先順位 (おおきい方が先に結合) があり、その優先順位と結合規則 (右結合、左結合、なし) を**演算子を定義しているソースコード内で指定することができます**。しかも通常の関数を二項

演算子として扱う[*11]こともでき、括弧なしで非常に複雑な式が記述可能です。パターンについても、関数にもなっているコンストラクタを二項演算子として使って、二項演算式の形式でパターンを記述していけます。もちろん二項演算子として使った場合のコンストラクタにも優先順位と結合規則があり、指定の方法は同様となっています。

ここで問題となるのは**構文解析時に優先順位が決定していない**ということです。ここでは expression、パターン共に、引数と演算子が交互に連なっているリストとして構文解析を行なっています。

```
/* parser.mly 二項演算 expression */
... /* 略 */
/* expression */
exp:
  exp0  { E.Top ($1, None) }
| exp0 KS_2_COLON context KS_R_W_ARROW typ  { E.Top ($1, Some ($5, Some $3)) }  /*(expression type signature)*/
| exp0 KS_2_COLON typ  { E.Top ($1, Some ($3, None)) }  /*(expression type signature)*/

/*
lexp6:
  - exp7
;
*/

/*
expi    ->      expi+1 [qop(n,i) expi+1]
        |       lexpi
        |       rexpi
lexpi   ->      (lexpi | expi+1) qop(l,i) expi+1
rexpi   ->      expi+1 qop(r,i) (rexpi | expi+1)
*/

/*
exp0:   ->      [-] exp10 {qop [-] exp10}
*/

exp0:
  op2_expn_list  { E.Exp0 $1 }

op2_expn_list:
  ks_minus exp10 op2_right_section  { E.ExpF (E.Minus $2, $3) }
| exp10 op2_right_section  { E.ExpF ($1, $2) }
| ks_minus exp10  { E.ExpF (E.Minus $2, E.Op2End) }
| exp10  { E.ExpF ($1, E.Op2End) }

op2_right_section:
  qop op2_expn_list { E.Op2F ($1, $2) }

op2_left_section:
  ks_minus exp10 qop op2_left_section  { E.ExpF (E.Minus $2, E.Op2F ($3, $4)) }
| exp10 qop op2_left_section  { E.ExpF ($1, E.Op2F($2, $3)) }
| ks_minus exp10 qop  { E.ExpF (E.Minus $2, E.Op2F ($3, E.Op2NoArg)) }
| exp10 qop  { E.ExpF ($1, E.Op2F ($2, E.Op2NoArg)) }

exp10:
  KS_B_SLASH apat_list KS_R_ARROW exp  { E.LambdaE ($2, $4) }   /*(lambda abstraction, n>=1)*/
| K_LET decl_list K_IN exp  { E.LetE ($2, $4) }         /*(let expression)*/
| K_IF exp K_THEN exp K_ELSE exp  { E.IfE ($2, $4, $6) }        /*(conditional)*/
| K_CASE exp K_OF SP_LEFT_BRACE alt_list SP_RIGHT_BRACE  { E.CaseE ($2, $5) }   /*(case expression)*/
| K_DO SP_LEFT_BRACE stmt_list_exp SP_RIGHT_BRACE  { E.DoE $3 }         /*(do expression)*/
| fexp  { E.FexpE $1 }

/*
 fexp    ->      [fexp] aexp     (function application)
*/

fexp:
  aexp_list  { E.make_fexp $1 }
;

aexp_list:
  aexp aexp_list  { fun fexp -> $2 (E.FappE (fexp, $1)) }
| aexp  { fun fexp -> E.FappE (fexp, $1) }
;
/* fexp -- FfunE (fexp) */
/* fexp ae1 -- FappE (FfunE (fexp), ae1) */
/* fexp ae1 ae2 -- FappE (FappE (FfunE (fexp), ae1), ae2) */
```

[*11] バッククオート (‘) でくくる

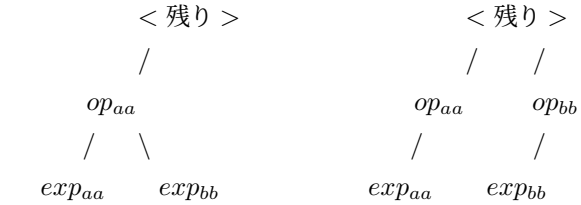
```
/* parser.mly パターンおよび 二項演算パターン */
.../* 略 */
pat:
  var ks_plus integer   /*(successor pattern)*/
      { match $3 with (S.Int (i), loc) -> P.PlusP($1, i, loc) | _ -> failwith "plus integer pattern syntax error." }
| pat0  { $1 }

/*
pati     ->      pati+1 [qconop(n,i) pati+1]
         |       lpati
         |       rpati
*/

/*
lpati   ->      (lpati | pati+1) qconop(l,i) pati+1
*/

/*
lpat6:
  ks_minus integer      (negative literal)
      { match $2 with
          (S.Int (v), l) -> S.P.MIntP (v, l)
        | _ -> failwith "negative integer literal pattern syntax error." }
| ks_minus float        (negative literal)
      { match $2 with
          (S.Float (v), l) -> S.P.MFloatP (v, l)
        | _ -> failwith "negative integer literal pattern syntax error." }
;
*/

/*
rpati   ->      pati+1 qconop(r,i) (rpati | pati+1)
*/

pat0:
  op2_patn_list  { P.Pat0 $1 }

op2_patn_list:
  ks_minus integer op2_patn_right
    { let p = match $2 with
        (S.Int (x), loc) -> P.MIntP (x, loc)
      | _ -> failwith "negative integer literal pattern syntax error."
      in P.PatF (p, $3)
    }
| ks_minus float op2_patn_right
    { let p = match $2 with
        (S.Float (x), loc) -> P.MFloatP (x, loc)
      | _ -> failwith "negative integer literal pattern syntax error."
      in P.PatF (p, $3)
    }
| pat10 op2_patn_right  { P.PatF ($1, $2) }

op2_patn_right:
  qconop op2_patn_list  { P.Op2F ($1, $2) }
|   { P.Op2End }

pat10:
  apat  { $1 }
| gcon apat_list        /*(arity gcon = k, k>=1)*/
      { P.ConP($1, $2) }

apat_list:
  apat apat_list { $1::$2 }
| apat           { [$1] }

apat:
  var
      { P.VarP $1 }
| var KS_AT apat        /*(as pattern)*/
      { P.AsP($1, $3) }
| gcon  /*(arity gcon = 0)*/
      { P.ConP($1, []) }
| qcon SP_LEFT_BRACE fpat_list SP_RIGHT_BRACE   /*(labeled pattern, k>=0)*/ /* may be error pattern */
      { P.LabelP($1, $3) }
| literal
      { P.LiteralP($1) }
| K_WILDCARD    /*(wildcard)*/
      { P.WCardP }
| SP_LEFT_PAREN pat SP_RIGHT_PAREN      /*(parenthesized pattern)*/
      { $2 }
| SP_LEFT_PAREN tuple_pat SP_RIGHT_PAREN        /*(tuple pattern, k>=2)*/
      { P.TupleP $2 }
| SP_LEFT_BRACKET list_pat SP_RIGHT_BRACKET     /*(list pattern, k>=1)*/
      { P.ListP $2 }
| KS_TILDE apat         /*(irrefutable pattern)*/
      { P.Irref $2 }
```

■二項演算子の優先順位の解決

一度、構文解析が完了した後に二項演算子の優先順位の解決を行ないます。構文木を再帰的に辿り、二項演算式の交互のリストを木構造に置き換えます。

具体的には、二項演算式の一番左側の 2 つの単項表現 exp_{aa}, exp_{bb} および 2 つの演算子 op_{aa}, op_{bb} について着目する

と、二項演算式は次の二通りのいずれかの形になっているはずです。



この考え方をもとに演算子の優先順位と結合規則を考慮しつつ再帰呼び出しの関数を実装すると以下のようになります。

```
  type 'exp op2list_opf =
      Op2F of (ID.idwl * 'exp op2list_expf)
    | Op2End
  and 'exp op2list_expf =
      ExpF of ('exp * 'exp op2list_opf)
(*    | UniOpF of (ID.idwl * 'exp * 'exp op2list_opf) *)
    | Op2NoArg
  ... (* 中略 *)
  let rec explist2term func list =
    let exp10_fun = SYA.maptree_exp10 func in

    let rec fold_leafs list =
      let scanned_op2exp op expAA expBB =
        E.VarOp2E (op,
                   exp10_fun expAA,
                   exp10_fun expBB) in
        match list with
          | E.ExpF (exp, E.Op2End) -> (* list *)
              E.uni_exp (exp10_fun exp)
          | E.ExpF (expAA, E.Op2F (op_aa,
                                   (E.ExpF (expBB, E.Op2End)))) ->
              E.uni_exp (scanned_op2exp op_aa expAA expBB)
          | E.ExpF (expAA, E.Op2F ((op_aa, _) as op_aa_wl,
                                   ((E.ExpF (expBB, E.Op2F ((op_bb, _) as op_bb_wl, rest))) as cdr))) ->
              begin
                let (aa_fixity, _) = eval_op2_fixity modbuf op_aa in
                let (bb_fixity, _) = eval_op2_fixity modbuf op_bb in
                  (* F.printf "(%s, %d) vs (%s, %d)\n" (ID.name_str op_aa) (snd aa_fixity) (ID.name_str op_bb) (snd bb_fixity); *)
                  match (aa_fixity, bb_fixity) with
                    | ((_, aa_i), (_, bb_i)) when aa_i > bb_i ->
                        fold_leafs (E.expf_cons (scanned_op2exp op_aa_wl expAA expBB) op_bb_wl rest)
                    | ((SYN.InfixLeft, aa_i), (SYN.InfixLeft, bb_i)) when aa_i = bb_i ->
                        fold_leafs (E.expf_cons (scanned_op2exp op_aa_wl expAA expBB) op_bb_wl rest)
                    | ((_, aa_i), (_, bb_i)) when aa_i < bb_i ->
                        E.expf_cons expAA op_aa_wl (fold_leafs cdr)
                    | ((SYN.InfixRight, aa_i), (SYN.InfixRight, bb_i)) when aa_i = bb_i ->
                        E.expf_cons expAA op_aa_wl (fold_leafs cdr)
                    | _ ->
                        failwith (F.sprintf "Syntax error for operator priority. left fixity %s, right fixity %s"
                                    (SYN.fixity_str aa_fixity)
                                    (SYN.fixity_str bb_fixity))
              end
          | _ -> failwith "Arity 2 operator expression syntax error."
    in
      match fold_leafs list with
        | E.ExpF (exp, E.Op2End) -> exp
        | E.ExpF (exp, E.Op2F (_, E.Op2NoArg)) -> failwith "explist2term: section not implemented."
        | folded -> explist2term func folded
```

```
type 'pat op2list_opf =
    Op2F of (ID.idwl * 'pat op2list_patf)
  | Op2End
and 'pat op2list_patf =
    PatF of ('pat * 'pat op2list_opf)
  | Op2NoArg
... (* 中略 *)
let rec patlist2term min_i func list =
  let pat_fun = SYA.maptree_pat func in

  let rec fold_leafs list =
    let scanned_op2pat op patAA patBB =
      P.ConOp2P (op,
                 pat_fun patAA,
                 pat_fun patBB) in

      match list with
        | P.PatF (pat, P.Op2End) ->
            P.uni_pat (pat_fun pat)
        | P.PatF (patAA, P.Op2F (op_aa_wl, (P.PatF (patBB, P.Op2End)))) ->
            P.uni_pat (scanned_op2pat op_aa_wl patAA patBB)
        | P.PatF (patAA, P.Op2F ((op_aa, _) as op_aa_wl, ((P.PatF (patBB, P.Op2F ((op_bb, _) as op_bb_wl, rest))) as cdr))) ->
            begin
              let (aa_fixity, _) = eval_op2_fixity modbuf op_aa in
              let (bb_fixity, _) = eval_op2_fixity modbuf op_bb in
                match (aa_fixity, bb_fixity) with
                    ((_, aa_i), _) when aa_i < min_i ->
                      failwith (F.sprintf "Pat%d cannot involve fixity %s operator." min_i (SYN.fixity_str aa_fixity))
                  | (_, (_, bb_i)) when bb_i < min_i ->
                      failwith (F.sprintf "Pat%d cannot involve fixity %s operator." min_i (SYN.fixity_str bb_fixity))
                  | ((_, aa_i), (_, bb_i)) when aa_i > bb_i ->
                      fold_leafs (P.patf_cons (scanned_op2pat op_aa_wl patAA patBB) op_bb_wl rest)
                  | ((SYN.InfixLeft, aa_i), (SYN.InfixLeft, bb_i)) when aa_i = bb_i ->
                      fold_leafs (P.patf_cons (scanned_op2pat op_aa_wl patAA patBB) op_bb_wl rest)
                  | ((_, aa_i), (_, bb_i)) when aa_i < bb_i ->
                      P.patf_cons patAA op_aa_wl (fold_leafs cdr)
                  | ((SYN.InfixRight, aa_i), (SYN.InfixRight, bb_i)) when aa_i = bb_i ->
                      P.patf_cons patAA op_aa_wl (fold_leafs cdr)
                  | _ ->
                      failwith (F.sprintf "Syntax error for operation priority. left fixity %s, right fixity %s"
                                  (SYN.fixity_str aa_fixity)
                                  (SYN.fixity_str bb_fixity))
            end
        | _ -> failwith "Arity 2 operator pattern syntax error."
  in
    match fold_leafs list with
      | P.PatF (pat, P.Op2End) -> pat
      | P.PatF (pat, P.Op2F (_, P.Op2NoArg)) -> failwith "patlist2term: section not implemented."
      | folded -> patlist2term min_i func folded
```

5.6 Haskell の評価器

5.2.1 節でも述べた、環境を渡してゆく方法で Haskell の評価器を実装するために以下の型の環境 (env_t)、単純な closure(lambda_t)、関数定義のための closure(closure_t) を定義しました。

先の議論での closure は仮引数リスト、 body の expression、および環境の 3 つを持っているデータ型でした。こちらで同じ役割を果たす lambda_t では、 Haskell の関数の仮引数がパターン照合 (pattern match) を行なうので仮引数リストの代わりに pattern のリストを持っているのと、 Haskell の関数束縛宣言およびパターン照合による束縛宣言における where 節の環境を構築するための関数を保持するのフィールドを増やしています。

また、 Haskell の関数定義の pattern match によって複数の expression を書き分ける機能を、単純な closure に置き換えるのは困難なため、複数の closure を持つことのできる型 closure_t を導入しました。

```
type lambda_t = {
  arg_pat_list : P.pat list;
  body : E.t;
  lambda_env : env_t;
  apply_where : (env_t -> env_t);
}

and closure_t =
  | SPat of (lambda_t)
  | MPat of (lambda_t list)
  | Prim of (thunk_t list -> value_t)

and value_t =
  | Bottom
  | IO
  | Literal of SYN.literal
  | Cons of (ID.id * (thunk_t list))
  | LabelCons of (ID.id * (ID.id, thunk_t) OH.t )
  | Tuple of (thunk_t list)
  | List of (thunk_t list)
  | Closure of (closure_t * int * E.aexp list)

and thunk_t = unit -> value_t

and pre_value_t =
    Thunk of (unit -> value_t)
  | Thawed of value_t

and scope_t = (S.t, thunk_t) H.t

(* あるスコープでの環境 *)
and env_t = {
  symtabs : (scope_t) list;
  top_scope : scope_t;
}
```

5.6.1 遅延評価

Haskell の評価戦略はデフォルトで遅延評価 (lazy evaluation) です。次のようなプログラムを定義して g（f 1 2）2 を評価することを考えてみます。

```
f x y = x + y
g x y = x * y
```

話を簡単にするために +, * はプリミティブであるということにすると、まず g を評価すると（f 1 2）* 2 となります。つぎに、*はそれ以上評価しても意味がないプリミティブなので f が評価され、（1 + 2）* 2 となり、以下 3 * 2, 6 となって評価が終了します。

他の多数のプログラミング言語は eager evaluation を採用しているものが多く、その場合は関数の引数が完全に評価されたあとに関数が評価されます。書きくだしてみると、g（f 1 2）2 − > g（1 + 2）2 − > g 3 2 − > g 3 2 − > 3 * 2 − > 6 のような感じになるはずです。

■遅延評価の実装

lazy evaluation を実装するには、関数の引数を評価するときに最後まで評価するのではなく、引数の計算を行なうような closure を生成することで対応することができます。この小さな closure をここでは thunk と呼んでいます。環境 env_t の持っている値を thunk_t にしているのはそのためです。

make_thunk で thunk ごとに評価前の関数 (Thunk) または評価後の値 (Thawed) を保持する pre_value_t 型の構造を作っています。 thunk を初めて呼び出したときに評価が行なわれて値が保持され、以降の thunk 呼び出しでは単に Thawed の保持する値が返るようになります。

```
and thunk_t = unit -> value_t

and pre_value_t =
    Thunk of (unit -> value_t)
  | Thawed of value_t

and scope_t = (S.t, thunk_t) H.t

(* あるスコープでの環境 *)
and env_t = {
  symtabs : (scope_t) list;
  top_scope : scope_t;
}
...(* 中略 *)
let thunk_value thunk =
  match thunk with
      Thunk (f) -> f ()
    | Thawed (v) -> v

let expand_thunk thunk_ref =
  match !thunk_ref with
      Thunk (_)  ->
        let v = thunk_value (!thunk_ref) in
        let _ = thunk_ref := Thawed v in
          v
    | Thawed (v) -> v

let make_thawed value =
  (fun () -> value)

let make_thunk eval_fun env evalee =
  let delay_fun = fun () -> (eval_fun env evalee) in
  let thunk_ref = ref (Thunk delay_fun) in
    fun () -> expand_thunk thunk_ref
```

5.6.2 遅延パターン照合

Haskell のパターン照合 (pattern match) は lazy evaluation と組み合わさるようにして動作します。実際の評価に必要な部分しか pattern match に対応する expression を評価しないように動きます。

次のプログラムを考えます。

```
main = let { (p, (q, r)) = (print 1, (print 2, print 3)) } in
       q
```

このプログラムを実行すると 2 のみが出力されます。`(p, (q, r))` と `(print 1, (print 2, print 3))` の pattern match が行なわれますが実際に必要になる `q` のみが最後まで評価されます。

■遅延パターン照合の実装

pattern match の際に pattern に従って構造分解を行ない、その構造分解に対応した thunk から thunk への分解を行なっていきます。末端に変数の pattern があれば環境に thunk を書き込みます。 pattern match の失敗をたとえば case 式で認識するために真理値を返しています。

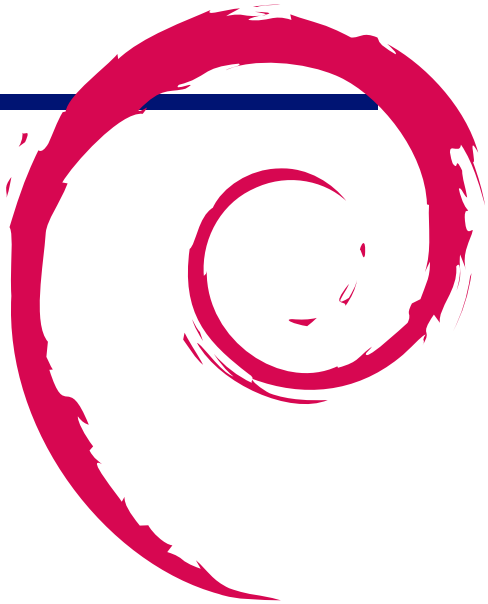
たとえばタプルの場合は、まずタプルに対応するはずの thunk を評価し、結果をタプルの要素に分解します。タプルのそれぞれの要素はやはりまた thunk になっているので、タプルの pattern のそれぞれの要素と pattern match を行なうように再帰します。それぞれの要素の pattern match が全て成功すれば、タプルの pattern match も成功です。

```
(* Lazy pattern match against thunk *)
and bind_pat_with_thunk pat =
  let sub_patterns_match env pat_list thunk_list =
    L.fold_left2
      (fun (matchp_sum, tlist_sum) pat thunk ->
         let (matchp, tlist) = bind_pat_with_thunk pat env thunk in
           (matchp_sum & matchp, L.append tlist_sum tlist))
      (true, [])
      pat_list
      thunk_list
  in
    match pat with
... (* 中略 *)
      | P.VarP (id, _) ->
          (fun env thunk ->
             let _ = bind_thunk_to_env env id thunk in (true, [thunk]))

      | P.AsP ((id, _), pat) ->
          (fun env thunk ->
             let (_, (matchp, tlist)) = (bind_thunk_to_env env id thunk,
                                         bind_pat_with_thunk pat env thunk)
             in (matchp, thunk :: tlist))
... (* 中略 *)
      | P.WCardP ->
          (fun _ thunk -> (true, [thunk]))

      | P.TupleP pat_list ->
          (fun env thunk ->
             let value = thunk () in
               match value with
                   Tuple (args) when (L.length args) = (L.length pat_list)
                     -> sub_patterns_match env pat_list args
                 | _ -> (false, [thunk]))

      | P.ListP pat_list ->
          (fun env thunk ->
             let value = thunk () in
               match value with
                   List (args) when (L.length args) = (L.length pat_list)
                     -> sub_patterns_match env pat_list args
                 | _ -> (false, [thunk]))
... (* 略 *)
```

5.7 まとめと今後の課題

結構な分量になってしまいましたが、それでもかなり端折って関数型言語のプログラミングと Haskell ライクなインタプリタを実装するにあたって苦心したトピックを紹介してみました。

今後の課題としては未実装の部分を実装していくことと、 ocamlyacc による parsing を Packrat parsing に置き換えることでより仕様に沿った構文解析をエレガントに行なえるようにすることです。



6 ブート方法が変わるよ

まえだこうへい

6.1 Squeeze からブート方法が変わる

Debian のブートの仕組みには、System-V 系 Unix では伝統的な init (sysvinit) が使われています。Debian の次期安定版 6.0 (コードネーム Squeeze) から、これが upstart に変わる予定です。今年の 3 月にフリーズ予定、8 月にリリース予定の Squeeze での予習の意味を込めて、今回はこの upstart に変わることになった背景、init との比較を含めた upstart の仕組み、実際に切り替え方法について説明します。

6.1.1 背景

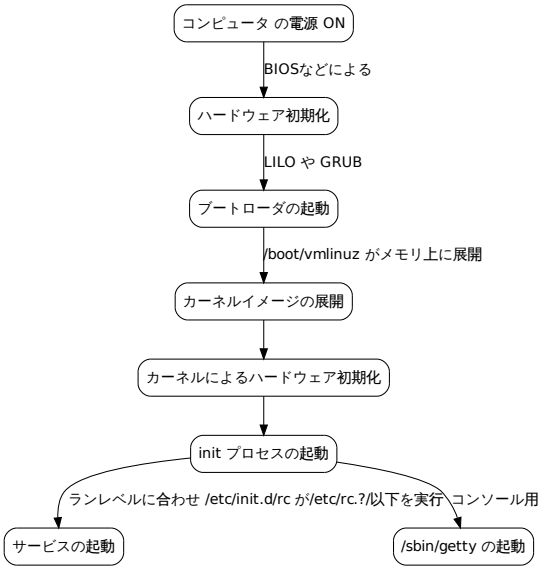
upstart は、Debian をベースとしたディストリビューションである、Ubuntu 独自に sysvinit の後継の仕組みとして開発されました。元々 sysvinit は伝統的で安定した仕組みではありますが、現在使われているハードウェアで使うには機能面、性能面から問題が出てきています。そこで、sysvinit の後継の仕組みとして、upstart 以外にも複数のブートプロセスの仕組みが開発されています。しかし、Ubuntu 以外にも Fedora 、また Google の Chrome OS, Chromium OS でも upstart が採用されています。Debian でも Squeeze から採用される予定ということもあり、sysvinit の後継は upstart に落ち着く可能性が高いのではないでしょうか。

6.1.2 そもそも init って何よ?

upstart の話をする前に、init って何? という人向けにまず説明しましょう。init は今更説明しなくても知っとるがな、という読者は、先に読み進めてください。

init は Unix/Linux システムにおいて、カーネルがブートした後、ユーザプログラムが起動するための仕組みです。例えば、あなたの使っているノート PC やサーバに入っている Debian システムでもこの仕組みが必ず入っています。マシンに電源を入れてから、ログイン画面が表示されるまでの流れは大まかには次のようになります。

図 2　ブートの流れ



init プロセスが起動すると、 init は /etc/inittab の内容に従って、プロセスのを生成や停止を行います。 inittab の書式は次のようになります。

```
id:runlevels:action:process
```

プロセスの生成に関わる部分には以下のようなものがあります。

```
si::sysinit:/etc/init.d/rcS

l1:1:wait:/etc/init.d/rc 1
l2:2:wait:/etc/init.d/rc 2
l3:3:wait:/etc/init.d/rc 3
l4:4:wait:/etc/init.d/rc 4
l5:5:wait:/etc/init.d/rc 5
```

一行目にランレベルの指定が無いのは、 action に sysinit が指定されているためです。これはステムブート中に実行され、他のブート用の action よりも優先して実行されます。/etc/init.d/rcS では

```
exec /etc/init.d/rc S
```

だけが実行されます。これは /etc/init.d/rc でのブート用の変数を設定します。この後、上記の rc スクリプトに対し、起動時に指定するランレベルを 引数として実行されますが、 Debian でのデフォルトは、ランレベル 2 で起動されます。

```
id:2:initdefault:
```

なので、実際には下記が実行されます。

```
l2:2:wait:/etc/init.d/rc 2
```

これにより /etc/rc2.d 以下のファイル名が **SNN** で始まるスクリプトが **NN** の二桁の数字の部分が" **昇順となるように一つずつ実行**" されます。これが起動が遅くなる原因の一つにもなっています。ただし、同じレベルのスクリプトは並行して実行されるようにはなっています。

```
# Now run the START scripts for this runlevel.
# Run all scripts with the same level in parallel
CURLEVEL=""
for s in /etc/rc$runlevel.d/S*
do
        # Extract order value from symlink
        level=${s#/etc/rc$runlevel.d/S}
        level=${level%%[a-zA-Z]*}
        if [ "$level" = "$CURLEVEL" ]
        then
                continue
        fi
        CURLEVEL=$level
        SCRIPTS=""
        for i in /etc/rc$runlevel.d/S$level*
        do
                [ ! -f $i ] && continue

                suffix=${i#/etc/rc$runlevel.d/S[0-9][0-9]}
                if [ "$previous" != N ]
                then
                        #
                        # Find start script in previous runlevel and
                        # stop script in this runlevel.
                        #
                        stop=/etc/rc$runlevel.d/K[0-9][0-9]$suffix
                        previous_start=/etc/rc$previous.d/S[0-9][0-9]$suffix
                        #
                        # If there is a start script in the previous level
                        # and _no_ stop script in this level, we don't
                        # have to re-start the service.
                        #
                        if [ start = "$ACTION" ] ; then
                                [ -f $previous_start ] && [ ! -f $stop ] && continue
                        else
                                # Workaround for the special
                                # handling of runlevels 0 and 6.
                                previous_stop=/etc/rc$previous.d/K[0-9][0-9]$suffix
                                #
                                # If there is a stop script in the previous level
                                # and _no_ start script there, we don't
                                # have to re-stop the service.
                                #
                                [ -f $previous_stop ] && [ ! -f $previous_start ] && continue
                        fi

                fi
                SCRIPTS="$SCRIPTS $i"
                if is_splash_stop_scripts "$suffix" ; then
                        $debug splash_stop || true
                fi
        done
        startup $ACTION $SCRIPTS
done
```

起動スクリプトの起動以外には、ランレベル 2 から 5 または、 2 か 3 の時にはコンソールから getty が実行されます。 action が `respawn` となっていますが、これは getty プログラムが終了したら、 init が再起動させるための指示です。あるユーザがコンソールからログインしたセッションを、ログアウトすると getty は終了しますが、 init により再びログイン画面で待ち受けることができる、というわけです。

```
1:2345:respawn:/sbin/getty 38400 tty1
2:23:respawn:/sbin/getty 38400 tty2
3:23:respawn:/sbin/getty 38400 tty3
4:23:respawn:/sbin/getty 38400 tty4
5:23:respawn:/sbin/getty 38400 tty5
6:23:respawn:/sbin/getty 38400 tty6
```

init の他の役割としては、システム停止時のプロセスの停止にも関わっています。

6.2 upstart とは

それでは、 init についての予備知識を得たところで、本題の upstart に入りましょう。 README にも記述されている upstart の主な特徴は次の 6 つです。

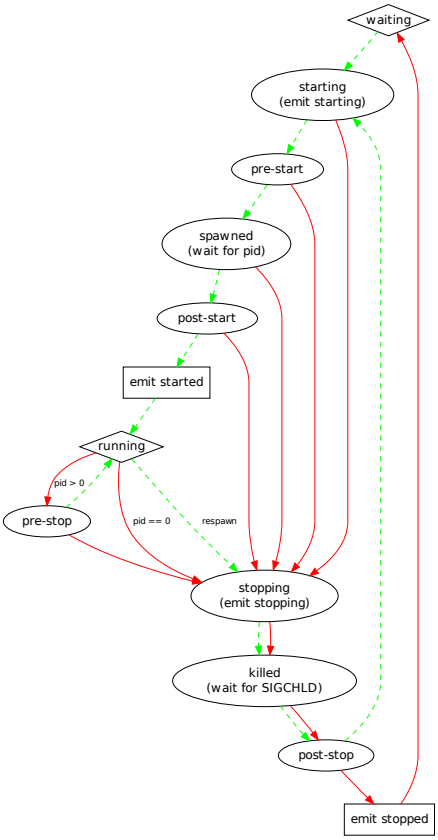
- イベントドリブンでタスクやサービスを起動・停止する。
- タスクやサービスが起動・停止することでイベントが発生する。
- イベントはシステム上の他のプロセスから受け取ることができる。
- サービスが予期せず突然終了しても再起動することができる。
- デーモンの監視と再起動は親プロセスから分離できる。

- D-Bus を通じて init デーモンと通信できる。

sysvinit とは異なり、イベントドリブンで非同期にタスクやサービスが起動される点が一番大きな違いでしょう。ただし、現在、 Squeeze/Sid で採用されている upstart は、 sysvinit の互換モードのものです。 upstart の最終目標は、イベントドリブンのブートプロセスに完全に移行することですが、互換モードでは、 sysvinit の動作を模倣しています。[*12]

ma upstart の状態遷移は次の図のようになります。[*13]

図 3　upstart 状態遷移



6.2.1　sysvinit と変わらない点

Debian の upstart は、前述のとおり、 Debian システムの起動･停止という重要な部分の置き換えを行うため互換モードのものが採用されています、下記は sysvinit と仕様上の変更がない部分です。なお、この 6.2.1 と次の 6.2.2 は upstart パッケージの README.Debian.gz に記載されている FAQ をまとめ直したものです。

- initscript がインストールされる場所。パスは /etc/init.d/
- 起動･停止用の initscript 。/etc/init.d/以下から /etc/rc?.d/ 以下に symlink が張られています。
- 起動･停止用 initscript の順序。 SNNname, KNNname として symlink を張る。 NN は 00 から 99。 K スクリプトが最初に序数順に実行され、 S スクリプトがそのあと実行されます。
- 現在および一つ前のランレベルを確認する方法。 `runlevel` コマンドを使います。
- ランレベルの変更方法。 `telinit` コマンドか `init` コマンドを実行します。
- デフォルトのランレベルの変更方法。/etc/inittab ファイルの `id:N:initdefault:` の N を書き換えます。

[*12] なお、 Ubuntu 9.10 以降では ネイティブモードに移行しているそうです。
[*13] upstart のドキュメントに付属のものを掲載。

- シャットダウンの方法。 upstart パッケージで提供される、 `shutdown` コマンドや `reboot`, `halt`, `poweroff` といったショートカットを使います。コンソールで Control-Alt-Delete を押してリブート出来る点も。
- シングルユーザモードにする方法。 GRUB から **(recoveryode)** オプションを選択か、カーネルのコマンドラインで、`-s`, `S`, `single` などの引数を指定する。稼働中のマシンでは、 `telinit 1` か `shutdown now` コマンドを実行します。

6.2.2 sysvinit と違う点

互換モードでも全てが sysvinit と動作が同じというわけではなく、 upstart の固有の部分もあります。主に getty 関連の設定が変わる点が大きな違いです。

- Control-Alt-Delete による挙動の変更方法。/etc/init/control-alt-delete.conf の **exec** で始まる行を変更します。キーを押しても何も実行されないようにするには、ファイルを消すだけです。

```
(snip)
start on control-alt-delete

task
exec shutdown -r now "Control-Alt-Delete pressed"
```

- getty を常駐させる数を減らす方法。/etc/init/ttyN.conf というファイルを変更します[*14]。必要なければファイルを削除します。

```
# tty1 - getty
#
# This service maintains a getty on tty1 from the point the system is
# started until it is shut down again.

description     "Start getty on tty1"
author          "Scott James Remnant <scott@netsplit.com>"

start on stopped rc RUNLEVEL=[2345]
stop on runlevel [!2345]

respawn
exec /sbin/getty 38400 tty1
```

- getty の設定変更の反映方法。ファイルを変更したり削除してもすぐには反映されません。停止は `stop ttyN` コマンドを、起動は `star ttyN` コマンドを実行します。
- getty のパラメータの変更方法。/etc/init/ttyN.conf の **respawn** で始まる行を変更します。
- getty を実行するランレベルのを変更方法。/etc/init/ttyN.conf の次の 2 行を変更します。 stop の**!** は否定で、start と stop の設定は、**!**以外は基本同じにします。
- getty の数を増やす方法。/etc/init/ttyN.conf を、 ttyS0 などの名前でコピーします。 **respawn** の行に必要な設定をを記述します。
- シリアルコンソールを追加する場合は、上記の”getty の数を増やす方法” と同じです。
- upstart が動かない場合のデバッグ方法。カーネルコマンドラインに`--debug` オプションをつけ、 `quiet` と `splash` オプションがある場合はそれらを削除します。 upstart が実行されるとデバッグメッセージが出力されます。[*15]。
- upstart が動かない場合のシステム復旧手順
  1. カーネルコマンドラインから、 `quite` と `splash` があればを削除し、 `init=/bin/bash` を引数で渡し起動すると、 root shell が起動されます。
  2. /etc/init.d/rcS を実行して、ハードウェアやネットワークの基本設定を行います。
  3. upstart がちゃんとインストールされているか確認します。/etc/init ディレクトリに全てのファイルがインストールされているかチェックし、正常にインストールされてない場合は upstart パッケージを再インストール

[*14] **N** は 1 から 6 の数字

[*15] initramfs-tools ではなく initramfs 生成ツールを使っている場合にはこのオプションを使うと既知のバグもあるので気をましょう。

します。

4. /etc/init にファイルが今度はちゃんとあるか確認します。
5. sync と reboot -f コマンドを実行し、マシンを再起動します。

- upstart ジョブリストをクエリする方法。 initctl list コマンドでジョブとステータスを表示します。

```
$ sudo initctl list
tty4 start/running, process 25474
rc stop/waiting
tty5 start/running, process 25478
control-alt-delete stop/waiting
rcS stop/waiting
rc-sysinit stop/waiting
dbus-reconnect stop/waiting
tty2 start/running, process 25473
tty3 start/running, process 25475
tty1 start/running, process 25477
tty6 start/running, process 25476
```

- ジョブの起動・停止方法。 start JOB, stop JOB コマンドを実行します。
- ジョブのステータス表示方法。 status JOB コマンド。

```
$ sudo status tty1
tty1 start/running, process 25477
```

- 手動でイベントを発行する方法。 initctl emit EVENT コマンドで名前付きイベントを発行し、待機中のジョブが状況に応じて起動 or 停止します。

6.3 upstart への切り替え

それでは、早速 sysvinit から upstart へ切り替えてみましょう。 Squeeze/Sid と Lenny との場合を見てみます。

6.3.1 Squeeze/Sid での場合

Squeeze/Sid での upstart への切り替えには、 upstart パッケージをインストールします。通常のパッケージのインストールとは異なり、続行する場合は、 **Yes, do as I say** と入力しなさい、というメッセージが表示されます。これは入れ替えは非常にリスクが高いためです。

```
$ sudo apt-get install upstart
パッケージリストを読み込んでいます... 完了
依存関係ツリーを作成しています
状態情報を読み取っています... 完了
以下の特別パッケージがインストールされます:
  dbus libdbus-1-3 libexpat1
提案パッケージ:
  dbus-x11
以下のパッケージは「削除」されます:
  sysvinit
以下のパッケージが新たにインストールされます:
  dbus libdbus-1-3 libexpat1 upstart
警告: 以下の不可欠パッケージが削除されます。
何をしようとしているか本当にわかっていない場合は、実行してはいけません!
  sysvinit
アップグレード: 0 個、新規インストール: 4 個、削除: 1 個、保留: 9 個。
1,005kB のアーカイブを取得する必要があります。
この操作後に追加で 2,105kB のディスク容量が消費されます。
重大な問題を引き起こす可能性のあることをしようとしています。
続行するには、'Yes, do as I say!' というフレーズをタイプしてください。
 ?] Yes, do as I say!
```

lxc の環境で試してみましたが、 getty がうまく動かず、起動しては突然死して、再起動されて、また突然死、というのを繰り返してしまうので、コンソールからのログインは出来ない状態です。[*16]

[*16] 2010 年 2 月 9 日現在

```
$ sudo lxc-start -n bootsid
cat: /proc/cmdline: No such file or directory
Setting the system clock.
Cannot access the Hardware Clock via any known method.
Use the --debug option to see the details of our search for an access method.
Unable to set System Clock to: Tue Feb 9 14:16:26 UTC 2010 ... (warning).
Activating swap...done.
mount: you must specify the filesystem type
Cannot check root file system because it is not mounted read-only. ... failed!
Setting the system clock.
Cannot access the Hardware Clock via any known method.
Use the --debug option to see the details of our search for an access method.
Unable to set System Clock to: Tue Feb 9 14:16:27 UTC 2010 ... (warning).
Cleaning up ifupdown....
Checking file systems...fsck from util-linux-ng 2.16.2
done.
Setting up networking....
Mounting local filesystems...done.
Activating swapfile swap...done.
Cleaning up temporary files....
Configuring network interfaces...done.
Setting kernel variables ...done.
Cleaning up temporary files....
Starting system message bus: dbus.
Starting OpenBSD Secure Shell server: sshd.
init: tty4 main process (239) terminated with status 1
init: tty4 main process ended, respawning
init: tty5 main process (241) terminated with status 1
init: tty5 main process ended, respawning
init: tty2 main process (242) terminated with status 1
init: tty2 main process ended, respawning
init: tty3 main process (244) terminated with status 1
init: tty3 main process ended, respawning
init: tty6 main process (245) terminated with status 1
init: tty6 main process ended, respawning
init: tty1 main process (306) terminated with status 1
init: tty1 main process ended, respawning
init: tty4 main process (307) terminated with status 1
init: tty4 main process ended, respawning
(snip)
```

ただし、 ssh 経由のターミナルログインは問題なくできました。

```
$ ssh bootsid
Enter passphrase for key '/home/user/.ssh/id_rsa':
Linux bootsid 2.6.32 #1 SMP Mon Dec 7 05:27:50 UTC 2009 x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Feb  9 14:18:38 2010 from 192.168.189.114
user@bootsid:~$
```

/etc/inittab の getty のエントリが残っているので不具合が起きているのか? とも思いましたが、それは原因ではありませんでした。コンソールからログイン出来ないことを除けば一応使えるようです。 KVM/QEMU などの環境で検証しなおしてみた方が良さそうです。

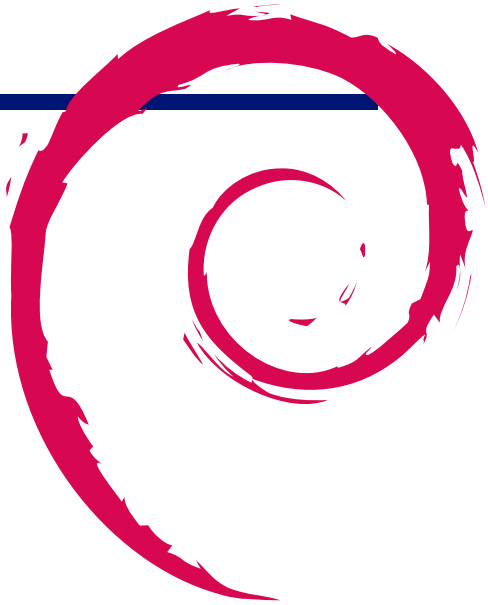
6.3.2 Lenny での場合

Squeeze/Sid でもうまく行っていない状況ですので、 Squeeze が stable としてリリースされるときに、検討しましょう[*17]。あるいは、 Sid にアップグレードすることを検討しても良いでしょう。

6.3.3 まとめ

現時点では、問題も抱えているようで、すんなり問題なく使うのは困難ではありますが、実際に切り替わった際にはオペレーション上も多少変更があります。 Ubuntu 9.10 で採用されている ネイティブモードでは更にオペレーションも変わるようですので、今回はじめて upstart を知ったという方は、今回をきっかけにぜひ早めに使い方や仕組みを予習しておくと良いでしょう。

[*17] upstart が予定どおり Squeeze に含まれていることが前提ですが。



7 東京エリア Debian 勉強会予約システムの構想

上川 純一

7.1 背景

東京エリア Debian 勉強会では「 えんかい君」を予約システムとして利用していました。えんかい君はシンプルなユーザインタフェースで認証もなく、全員のメールアドレスと名前が閲覧でき、他人の登録を誰でも削除できるなど、利用者を信頼したモデルになっていました。後で立ち上がった関西では cotocoto を利用していました。 cotocoto は DFSG の観点では non-free なサービスです。

「 えんかい君」は YLUG などでも利用されていましたが、不便でした。東京では、「 えんかい君」の制限を回避するため、課題の提出をメール経由でやっていました。当初はフリーフォーマットのメールを LaTeX 形式に上川がバッチで変換する形式をとっており、のちに LaTeX のソースコードをメールで git format-patch で送るという運用になっていました。ただ、 Git で課題提出をしていても、マージが面倒という問題点がありました。

2009 年 12 月の勉強会登録には実験的に atnd を利用しました。 atnd は DFSG non-free なサービスですが、最近流行している勉強会等の予約システムです。

DFSG 準拠のアプリケーションのほうが望ましいが、「 えんかい君」ではうまく運用できないということと、アプリケーション自体はシンプルな問題であることが予想されたため、自前で勉強会予約システムを準備してみることにしました。

7.2 実装目標

Debian 勉強会の予約システムでは何が必要でしょうか。

- イベントの主催者が簡便に登録情報を設定することができること。
- イベントの主催者が事前課題を設定し、回答を簡単に収集することができること。
- イベントの主催者が簡単に参加人数を確認することができること。
- イベントの主催者が新規参加者の情報を迅速に確認できること。
- イベントの主催者が参加者に直接連絡がとれる手段があること。
- 参加者が簡単に事前課題もあわせて登録できること。
- 参加者がイベント参加をキャンセルする方法があること。
- 参加者が参加しているイベントを把握する方法があること。

他にもいろいろあるかもしれませんが、とりあえずこういうものを目標にしてやってみました。
そして、 DFSG Free であることが望ましいです。

7.3 開発環境の準備

7.3.1 App Engine Python SDK の準備

今回はウェブアプリケーションのフレームワークとして、 Python 版の Google App Engine を利用しました。開発環境を Debian GNU/Linux sid 上で準備する方法を紹介します。

まず、 Debian GNU/Linux sid の環境を用意します。

次に、 Google App Engine の Python 版の開発環境をダウンロードします。 Google App Engine のサイト[*18]にいって最新の SDK をダウンロードしてきます。

「 Linux/その他のプラットフォーム」向けの google_appengine_1.3.1.zip をダウンロードしてきました。

```
# apt-get install unzip python python-openssl python-webtest python-yaml
$ wget http://googleappengine.googlecode.com/files/google_appengine_1.3.1.zip
$ unzip google_appengine_1.3.1.zip
```

これでインストールは完了です。 Google App Engine のインストールディレクトリを ./google_appengine, App Engine アプリケーションのソースコードのおいている場所を ./utils/gae とします。 utils/gae ディレクトリにから dev_appserver.py を実行すれば、開発用のウェブサーバが起動します。

```
hoge@core2duo:appengine/utils/gae$ ../../google_appengine/dev_appserver.py .
INFO     2010-02-16 15:28:08,816 appengine_rpc.py:159] Server: appengine.google.com
Allow dev_appserver to check for updates on startup? (Y/n): n
dev_appserver will not check for updates on startup.  To change this setting, edit /home/hoge/.appcfg_nag
WARNING  2010-02-16 15:28:13,792 datastore_file_stub.py:623] Could not read datastore data from /tmp/dev_appserver.datastore
WARNING  2010-02-16 15:28:13,906 dev_appserver.py:3581] Could not initialize images API; you are likely missing the Python "PIL" module. Impo
INFO     2010-02-16 15:28:13,914 dev_appserver_main.py:399] Running application debianmeeting on port 8080: http://localhost:8080
```

7.3.2 テストの実行方法

Django の通常のアプリケーションはテスト用の仕組みがあるようなのですが、 appengine にはないようです。ここでは、 WebTest モジュールを利用して自動テストコードを実装しています。

```
$ PYTHONPATH=../../google_appengine:../../google_appengine/lib/django/ \
 python testSystem.py
```

7.4 実装

7.4.1 認証の仕組み

このアプリケーションでは Google App Engine を利用しています。ユーザ認証は Google App Engine で標準で提供される Google の認証を流用しています。パスワードの管理やユーザのメールアドレスの管理などをフレームワークに一任

[*18] http://code.google.com/intl/ja/appengine/

することで管理を簡単にしています。

7.4.2 データベースの構造

バックエンドのデータベースには、 AppEngine の Datastore を利用しています。 Event と、 Attendance と UserRealName というのを定義しています。

Event は主催者がイベントについて登録した情報を保持しています。イベント毎に存在しています。

Attendance はユーザがイベントに登録したという情報を保持しています。イベントに対して登録したユーザの数だけ存在します。

UserRealname はユーザの表示名前の情報を保持しています。各ユーザ毎に存在します。

```
class Event(db.Model):
    eventid = db.StringProperty()
    owner = db.UserProperty() # the creator is the owner
    owners_email = db.StringListProperty() # allow owner emails to be added if possible
    title = db.StringProperty()
    location = db.StringProperty(multiline=True)
    content = db.StringProperty(multiline=True)
    content_url = db.StringProperty()
    prework = db.StringProperty(multiline=True)
    event_date = db.StringProperty()
    timestamp = db.DateTimeProperty(auto_now_add=True)
    capacity = db.IntegerProperty() # the number of possible people attending the meeting

class Attendance(db.Model):
    eventid = db.StringProperty()
    user = db.UserProperty()
    user_realname = db.StringProperty() # keep a cache of last realname entry.
    prework = db.StringProperty(multiline=True) # obsolete, but used in initial version
    prework_text = db.TextProperty() # Used everywhere, populate from prework if available.
    attend = db.BooleanProperty()
    enkai_attend = db.BooleanProperty()
    timestamp = db.DateTimeProperty(auto_now_add=True)

class UserRealname(db.Model):
    """Backup of user realname configuration so that user doesn't have to reenter that information."""
    user = db.UserProperty()
    realname = db.StringProperty()
    timestamp = db.DateTimeProperty(auto_now_add=True)
```

7.4.3 ソースコードの構造

ソースコードは現在下記の構成です。

- debianmeeting.py: どのページがどのコードを呼び出すのかという部分を管理しているコードです。あと、どこに入れるのか迷ったコードもここにあるかも。
- admin_event.py: 主催者のイベントの管理関連のコードです。
- user_registration.py: ユーザの登録関連のコードです。
- webapp_generic.py: とりあえず共通のロジックを定義しています。 POST と GET を同じように扱うためのコードなどが入っています。
- schema.py: データストアのスキーマが定義されています。
- send_notification.py: メール送信と XMPP 送信ロジックが記述されています。
- testSystem.py: ユニットテストです。

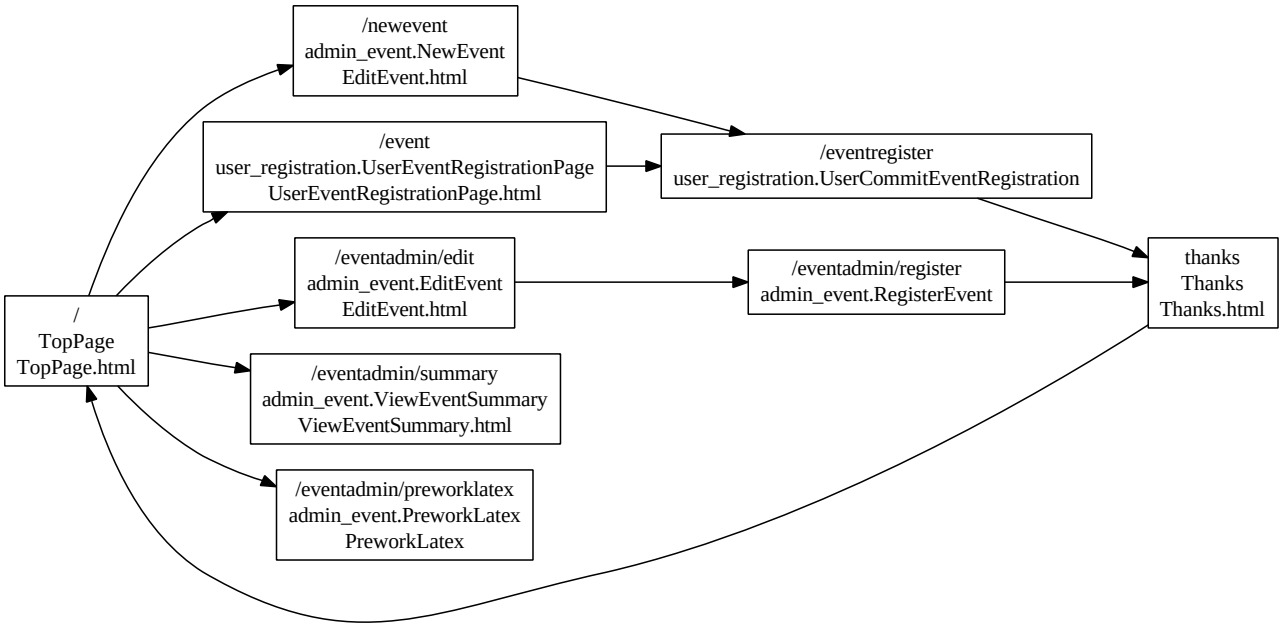
ソース内部からテンプレートファイルが参照されています。

- EditEvent.html
- PreworkLatex.txt
- RegisterEvent.txt
- Thanks.html
- TopPage.html
- UserCommitEventRegistration.txt
- UserEventRegistrationPage.html

- `UserEventRegistrationPage_Simple.html`
- `ViewEventSummary.html`

7.4.4 ウェブページの遷移

ウェブページの遷移とソースコードの対応をみてみます。



7.5 今後の展望

とりあえずは動いています。今後、何が変わるべきか。今後どういう点が実装されるべきか。パッチウェルカム。

Debian 勉強会資料

2010 年 2 月 20,21 日　　初版第 1 刷発行
東京エリア Debian 勉強会（ 編集・ 印刷・ 発行）