目次

第 0 章　まえがき **iii**

第 1 章　ATS 言語 string ライブラリ探訪 @master_q **1**
1.1 探検の前に 1
1.2 2 つの文字列ライブラリ 2
1.3 最初の地図を手に入れよう 5
1.4 string 型とは？ 6
1.5 string 型の値をコンソールに印字する 7
1.6 strchr 関数 8
1.7 string 型から strnptr 型へ 11
1.8 malloc_gc 関数の静的な意味 12
1.9 castvwtp_trans 関数とは何か 14
1.10 破壊的な文字列変更 15
1.11 strnptr 型から strptr 型へ 16
1.12 文字列の解放 17
1.13 その先へ 17
1.14 ライセンス 18
1.15 参考文献 18

第 2 章　Haskell で状態を扱う @dif_engine **19**
2.1 プロローグ 19
2.2 モナドとは何か 22
2.3 モナドを使って状態を持つ関数を扱う方法 35
2.4 エピローグ 39

第 3 章　【新編】加速しなイカ？【叛逆の物語】 @nushio **41**
3.1 プロローグ 41
3.2 夢のようなライブラリがあった、ような… 41
3.3 見かけはとっても簡潔だなって 43
3.4 並列度も、演算重複も、メモリ負荷も、大事だよ 45
3.5 本当のピーク性能と向き合えますか？ 49
3.6 見滝原中学校　Ninja 結界 51
3.7 Halide って、ほんと簡単 53
3.8 もう、Halide にも頼らない 55
3.9 見滝原駅前喫茶店 57
参考文献 58

第 **4** 章　ご注文は依存型ですか？ `@tanakh` **59**

会員名簿じゃなイカ**?** **68**

第 0 章

まえがき

関数型イカ娘とは!?

Q. 関数型イカ娘って何ですか?
A. いい質問ですね！
Q. 七冊目とか、そろそろあなた自身の本当の仕事と向き合えますか？
B. その気になれば痛みなんて完全に消しちゃえるんだ♪

関数型イカ娘とは、「イカ娘ちゃんは 2 本の手と 10 本の触手で人間どもの 6 倍の速度でコーディングが可能な超絶関数型プログラマー。型ありから型なしまでこよなく愛するが特に Scheme がお気に入り。」という妄想設定でゲソ。それ以上のことは特にないでゲソ。

この本は七冊目の関数型イカ娘の本でゲソ。シリーズも後半に突入したので、少し雰囲気を変えていくでゲソ! アニメ 3 期の放映が近いことを祈りつつ、関数型言語で地上を侵略しなイカ!

この本の構成について

この本は関数型とイカ娘のファンブックでゲソ。各著者が好きなことを書いた感じなので各章は独立して読めるでゲソ。以前の「λ カ娘」本がないと分からないこともないでゲソ。ただ、一般的な入門書ではないでゲソ。

第1章

ATS言語stringライブラリ探訪

— @master_q

1.1 探検の前に

ATS 言語 [*1] は依存型 (Dependent Types) と線形型 (Linear Types) をそなえたプログラミング言語で、そのコンパイルプロセスは図 1.1 のように C 言語を経由するんでゲソ。さらにフットプリントが小さく高速なバイナリを吐き出せるそうじゃなイカ。ランタイムがなく GC を無効にすることさえ可能なので、OS への依存度も小さいんでゲソ。実際、mbed マイコンプラットフォーム [*2] や 8-bit AVR を搭載した Arduino [*3] の上で ATS プログラムを動かした実績もあるんでゲソ。

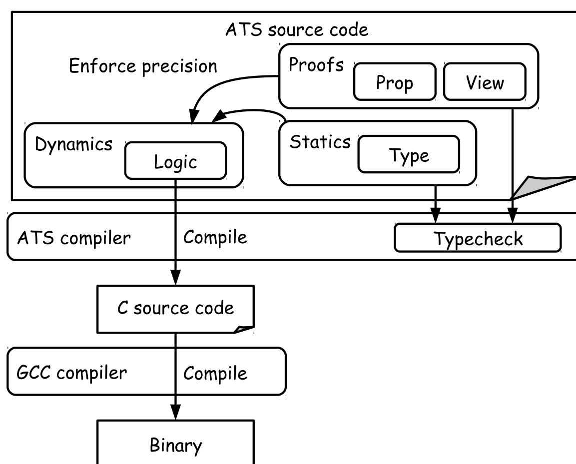

図 1.1: ATS のコンパイルプロセス

「ATS プログラミング入門 [*4]」という良い入門向けドキュメントはあるのでゲソが、これだけで依存型と線形型を駆使した ATS プログラミングができるようになるカ？ というと、それは少し厳しいとワシは思うのでゲソ。というのも依存型と線形型を通常の関数と混ぜて書くプログラミングスタイルは ATS 独自のもので、他のプログラミング言語の経験者にはなかなか習得しづらいんじゃなイカ？ そして、この混ぜて考えるということは 1 つの関数に静的な意味と動的な意味を同時に考えて与えるということなのでゲソ。とにかく馴染みのない概念でゲソ……

そこでこの記事では、他のプログラミング言語でも馴染み深い処理である「文字列処理」に焦点をしぼって、ATS 言語におけるプログラミングの作法を知る探検をしてみようと思うでゲソ。もしこの記事でわからない事柄があったら、まずは先の「ATS プログラミング入門」を (わからないなりに) 読んでみることをおすすめするでゲソ! 今回探検する ATS2 コンパイラのバージョンは ats2-positiats-0.1.0 でゲソ。

[*1] http://www.ats-lang.org/
[*2] https://github.com/fpiot/mbed-ats
[*3] https://github.com/fpiot/arduino-mega2560-ats
[*4] http://jats-ug.metasepi.org/doc/ATS2/INT2PROGINATS/index.html

1.2 2 つの文字列ライブラリ

ATS2 の基盤ライブラリである ATSLIB/prelude *5 には 2 つの文字列ライブラリがあるんでゲソ。それは ATSLIB/prelude/string *6 と ATSLIB/prelude/strptr *7 でゲソ。前者は不変の (immutable な) 文字列を、後者は線形型の文字列を扱うんでゲソ。

まずは簡単なプログラムを書いてみようじゃなイカ。

```
$ vi teststr.dats
#include "share/atspre_staload.hats"
implement main0 () = {
  val s1 = "Hello!"
  val () = print s1
  val () = print "\n"

  val pos = strchr (s1, 'o')
  val () = println! ("The 'o' is found at ", pos, "th in '", s1, "'")

  val s2 = string1_copy s1
  val () = s2[5] := '?'
  val s3 = strnptr2strptr s2
  val () = println! ("s1 := ", s1, " / s2 := ", s3)
  val () = free s3
}
$ patscc -DATS_MEMALLOC_LIBC -o teststr teststr.dats
--snip--
The 1st translation (fixity) of [teststr.dats] is successfully completed!
The 2nd translation (binding) of [teststr.dats] is successfully completed!
The 3rd translation (type-checking) of [teststr.dats] is successfully completed!
The 4th translation (type/proof-erasing) of [teststr.dats] is successfully compl
eted!
--snip--
$ ./teststr
Hello!
The 'o' is found at 4th in 'Hello!'
s1 := Hello! / s2 := Hello?
```

このプログラムは何をやっているのかというと、だいたいイカのような処理でゲソ。

1. "Hello!" という string 型文字列のリテラルを s1 に束縛
2. s1 を print 関数を使ってコンソールに印字
3. strchr 関数を使って s1 の何番目の文字に'o' が出現するか調べる
4. s1 の指す string 型の文字列を strnptr(n) 型の文字列にコピーして s2 という名前で束縛

*5 http://www.ats-lang.org/LIBRARY/#ATSLIB_prelude
*6 http://www.ats-lang.org/LIBRARY/prelude/SATS/DOCUGEN/HTML/string.html
*7 http://www.ats-lang.org/LIBRARY/prelude/SATS/DOCUGEN/HTML/strptr.html

5. s2 の 5 文字目を'?' に変更
6. s2 の指す strnptr(n) 型の文字列を strptr 型に変換して s3 という名前で束縛
7. s3 の strptr 型文字列を free 関数で解放

この teststr.dats プログラムは単なる ML もどき文法のプログラムに見えるでゲソが、実はちゃんと依存型と線形型を使っている、より安全なプログラムなのでゲソ。既に string ライブラリ側に依存型と線形型による強制が仕込まれているので、ライブラリを使うプログラマが依存型と線形型を意識しなくてもプログラミングができるのでゲソ。

ところで、なぜコンパイル時検査がうれしいのでゲソ？ テストを書くことによっても同様のエラーを実行時に検出できるのではなイカ？ コンパイル時検査と実行時検査とは少し似ているけれど違う手法なのでゲソ。テストは実行時にエラーを検出する手法でゲソ。契約プログラミングも実行時にエラーを検出するしくみでゲソ。一方で、依存型や線形型のような型を使ったコンパイル時検査はコンパイル時に静的なエラーを検出できるのでゲソ。

実行時検査であるテストのことを思い出してみるでゲソ。テストとは、プログラムのコンパイルが終わった後、実際にプログラムを実行することで検査を行なうことでゲソ。テストにはブラックボックステストとホワイトボックステストがあり、前者の方法であればプログラムの詳細を知らなくても実行時検査を行なうことができたでゲソ。しかし検査の網羅率であるテストカバレッジを向上させるためには時にホワイトボックステストを行なう必要があり、さらには結合テストではなくプログラムをモジュールに分割してテストを行なう必要もあったでゲソ。モジュールテストやユニットテストを行なう場合にはモジュールが依存する先をエミュレートするテストスタブを作る必要も生じることがあったでゲソ。これらのテスト戦略はプログラムを作る前に綿密に計画を立てる必要があったじゃなイカ。またテストは例示によって検査を行なうためテストカバレッジを向上させるのにはおのずと限界があったでゲソ。

一方、コンパイル時検査はプログラムの静的な性質そのものを検査できるのでゲソ! 簡単な例では「この関数の引数には NULL が入ってきてはならない」というような静的な性質は依存型によって強制することができるでゲソ。もう 1 つの簡単な例では、「このリソースは必ず hoge_close 関数によって解放されなくてはならない」というようなリソースに対する制約は線形型によって強制することができるんでゲソ。このような静的な性質はコンパイル前には意味論として扱うことができるのでゲソが、コンパイル後の実行時には意味論として見えなくなってしまうことが多いのでゲソ。

もちろん全ての不具合をコンパイル時に型によって検査できる訳ではないでゲソ。例えば外部からの入力がどのようなものであるかはコンパイル時には予測できないでゲソ。また内部でスレッドを使っている場合、スレッドとスレッドがどのように並列/並行実行されうるかもコンパイル時に予測することができないでゲソ。このような場合には静的に検証できるような特別な仕組みを作るか、やはりテストによって実行時検査を行なう必要があるのでゲソ。ここで強調しておきたいのは、実行時検査に限界があるのと同様に、コンパイル時検査で検出できる不具合の種類にも限界があるということでゲソ。

とはいえワシの思うところでは、型によるコンパイル時検査が使える場面では使うべきでゲソ。型による静的な性質の検査はいわば全通りテストのようなものなので、どのような実行時検査よりも網羅率が高くなるはずじゃなイカ。しかもその記述は性質そのものを表現しているので同等のテストと比較してもはるかに簡単でゲソ。そして、静的な検査で網羅できない範囲を動的な性質を扱えるテストでおぎなうのでゲソ。

さて、依存型と線形型によるコンパイル時検査の効果を確かめるために、少しいたずらをしてこのサンプルプログラムに不具合を混入させてみようと思うでゲソ。strnptr(n) 型の文字列 s2 の範囲外に文字を書き込んでみたのがソースコード teststr_e1.dats でゲソ。

```
$ vi teststr_e1.dats
#include "share/atspre_staload.hats"
implement main0 () = {
  val s1 = "Hello!"
  val s2 = string1_copy s1
  val () = s2[6] := '?' // <= Changed.
  val s3 = strnptr2strptr s2
  val () = println! ("s1 := ", s1, " / s2 := ", s3)
  val () = free s3
}
$ patscc -DATS_MEMALLOC_LIBC -o teststr_e1 teststr_e1.dats
--snip--
The 2nd translation (binding) of [teststr_e1.dats] is successfully completed!
/home/kiwamu/tmp/teststr_e1.dats: 119(line=5, offs=12) -- 124(line=5, offs=17):
 error(3): unsolved constraint: C3NSTRprop(main; S2Eapp(S2Ecst(<); S2EVar(4070
 ->S2Eintinf(6)), S2EVar(4069->S2Eint(6))))
typechecking has failed: there are some unsolved constraints: please inspect
 the above reported error message(s) for information.
```

すばらしいことにコンパイルエラーになったでゲソ。エラーメッセージは「“ $6 < 6$ ” という強制が main において解決できない」と言っているでゲソ。束縛名が見えないのでなんだかわけがわからないでゲソが、これは依存型のエラーでゲソ。後に調べるでゲソが、strnptr_set_at_gint 関数に割り当てられている全称量化の条件を “s2[6]” が満たしていないためにエラーになっているんじゃなイカ。

次のエラーは線形型のリソース s3 を解放しないままプログラムを終了してしまった例でゲソ。

```
$ vi teststr_e2.dats
#include "share/atspre_staload.hats"
implement main0 () = {
  val s1 = "Hello!"
  val s2 = string1_copy s1
  val () = s2[5] := '?'
  val s3 = strnptr2strptr s2
  val () = println! ("s1 := ", s1, " / s2 := ", s3)
//  val () = free s3 // <= Changed.
}
$ patscc -DATS_MEMALLOC_LIBC -o teststr_e2 teststr_e2.dats
--snip--
The 2nd translation (binding) of [teststr_e2.dats] is successfully completed!
/home/kiwamu/tmp/teststr_e2.dats: 59(line=2, offs=22) -- 250(line=9, offs=2):
 error(3): the linear dynamic variable [s3$3422(-1)] needs to be consumed but
 it is preserved with the type [S2Eapp(S2Ecst(strptr_addr_vtype); S2EVar(4075))]
 instead.
```

これもすばらしいことにコンパイル時エラーになったでゲソ! エラーメッセージは「線形の動的な値 s3 は消費 (consume) されるべきなのに、型 strptr_addr_vtype として残ってしまっている」と言っているでゲソ。これも後に調べるでゲソが線形型のリソースは生成したら必ずどこかで消費する必要があるのでゲソ。この線形型の特性によって strptr 型は文字列を格納したメモリ領域というリソースを安全に管理できるのでゲソ。

また一度解放したリソースを再度使おうとしてもエラーになるでゲソ。

```
$ vi teststr_e3.dats
#include "share/atspre_staload.hats"
implement main0 () = {
  val s1 = "Hello!"
  val s2 = string1_copy s1
  val () = s2[5] := '?'
  val s3 = strnptr2strptr s2
  val () = s2[5] := '=' // <= Changed.
  val () = println! ("s1 := ", s1, " / s2 := ", s3)
  val () = free s3
}
$ patscc -DATS_MEMALLOC_LIBC -o teststr_e3 teststr_e3.dats
--snip--
The 2nd translation (binding) of [teststr_e3.dats] is successfully completed!
/home/kiwamu/tmp/teststr_e3.dats: 172(line=7, offs=12) -- 175(line=7, offs=15):
 error(3): the linear dynamic variable [s2$3421(-1)] is no longer available.
```

このエラーメッセージは「線形の動的な値 s2 はもはや有効ではない」と言っているでゲソ。teststr_e3.dats のソースコードでは s2 を解放した後に使おうとしているでゲソ。strnptr2strptr 関数によって s2 は解放され、その直後の行で s2 の 5 文字目に'=' を上書きしようとしていて、この行でエラーが発生しているでゲソ。ちょっと注意してほしいのはここで言うリソースとはメモリリソースに限らない、ということでゲソ。生成されて消費されるものはなんであれリソースと見做すことができる可能性があるのでゲソ。

この記事では teststr.dats プログラムの動作を一つずつ追っていくことで、ATS プログラムの成り立ちを調べてみようと思うでゲソ。

1.3 最初の地図を手に入れよう

まずは ATS2 コンパイラのソースコードからライブラリに対応するソースコードの在処を確かめるでゲソ。

```
$ wget http://downloads.sourceforge.net/project/ats2-lang/ats2-lang/ats2-\
postiats-0.1.0/ATS2-Postiats-0.1.0.tgz
$ tar xf ATS2-Postiats-0.1.0.tgz
$ cd ATS2-Postiats-0.1.0
$ find prelude -name "string.*"
prelude/CATS/string.cats
prelude/DATS/string.dats
prelude/SATS/string.sats
```

第2章

Haskellで状態を扱う

— @dif_engine

物語の概要と目的

この章では、Haskell で「状態」を扱うための基本を学びます。いわゆる命令型言語では、変数に計算途中の値を格納しておき、必要に応じてその変数の値を書き換える（更新する）ことでプログラムを記述することができます。このようなプログラミングスタイルは、数学的なアルゴリズム記述との親和性が高く、多くのアルゴリズムが命令型言語で実装され、実用化されてきました。

一方、Haskell では一度定義した変数を書き換える方法は用意されていません。したがって、状態を積極的に使ったアルゴリズムを Haskell で使おうとするときには何らかの工夫をする必要があります。本稿ではその中で最も基本的なものの一つである「状態を取り回す」ためのハック — 関数型プログラミング界で「状態モナド」と呼ばれているもの — について説明します。

シンプルで汎用性の高いハックが大抵そうであるように、このハックにも理論的な基盤があります。状態モナドのデザインは、圏論の概念、特にモナドと結びついた Kleisli 圏と合わせて考えるとよく理解できます。本稿では Haskell をすでにある程度習得している読者を想定し、このデザインについて説明をします。圏論についてはとくに事前知識を要求しませんが、読者がある程度の数学的な素養、たとえば理工系の大学二年生に求められる程度の理解力を持っていることは想定します。

2.1 プロローグ

2.1.1 その『状態モナド』ってのを使えば参照透明性をぶっ壊せるんだろ？

「—— でもさ、Haskell の変数って参照透明でしょ？ つまり一度変数を定義したらその変数を書き換えたり出来ないわけだよね。そんなんじゃ内部状態を持つ関数とか書けそうにないけど」

さて、どう答えたものだろうか。

「不便かどうかは、主観的な話だからスルーかな。『内部状態を持つ関数』について言えば、状態モナドを使えばそういう関数を書けるし、参照透明性がそこまで重大な制約になるとは限らないわ」

「また『モナド』かー。てか Haskell ってなんでもモナドが出てくるのな」

「なんでもモナドというわけじゃないわ。モナドでは不十分な場面では、例えば Arrow なんかが使われることがあるし」

「まあそんなすごい話はともかく、まずモナドからして私はわかってないんだよなー。でさ、その『状態モナド』ってのを使えば参照透明性をぶっ壊せるんだろ？ なんかすごい話だよな」

「参照透明性をぶっ壊す —— ？ そんな物騒な事はしないわ。Haskell の基本的な性質はまった

(M1) $F(f) \circ \eta_X = \eta_Y \circ f \quad \left({}^\forall X, Y \in \mathrm{Ob}(\mathbf{C}), {}^\forall f \in \mathrm{Hom}(X,Y)\right).$
(M2) $F(f) \circ \mu_X = \mu_Y \circ F^2(f) \quad \left({}^\forall X, Y \in \mathrm{Ob}(\mathbf{C}), {}^\forall f \in \mathrm{Hom}(X,Y)\right).$
(M3) $\mu_X \circ F(\eta_X) = \mathrm{id}_{F(X)} \quad \left({}^\forall X \in \mathrm{Ob}(\mathbf{C})\right).$
(M4) $\mu_X \circ \eta_{F(X)} = \mathrm{id}_{F(X)} \quad \left({}^\forall X \in \mathrm{Ob}(\mathbf{C})\right).$
(M5) $\mu_X \circ F(\mu_X) = \mu_X \circ \mu_{F(X)} \quad \left({}^\forall X \in \mathrm{Ob}(\mathbf{C})\right).$

「な、なるほど……？？」

「順番に、**(M1)** は自然変換 η の条件 **(natx2)** を書き下したもので、**(M2)** は自然変換 μ の条件 **(natx2)** を書き下したもので、**(M3)** はモナドの条件 **(mon1)** を書き下したもので、**(M4)** はモナドの条件 **(mon2)** を書き下したもので、**(M5)** はモナドの条件 **(mon3)** を書き下したものになってるわ」

「なるほど」

Kleisli 圏

「モナドが半群の一般化だという話は、プログラミング言語への応用からすると脇道だと思う。モナドが役に立つ理由はなんといっても **Kleisli** 圏の存在だと思うわ」

「『くらいすり』は人の名前？」

「そうね [*2]。簡単に言えば、Kleisli 圏は、圏 $\mathbf{C}$ 上の自己関手 F のなかのこんな形の射を “合成” するような圏だと言えるわ」：

$$X \xrightarrow{f} F(Y).$$

「それって、普通に無理だよね。だって、図式を書いてみると

$$\begin{array}{ccccc} & & Y & \xrightarrow{g} & F(Z) \\ X & \xrightarrow[f]{} & F(Y) & & \end{array}$$

となって、矢印の始点と終点が合わないから合成できないし。待って、なんか思いついた。F は関手だから今の図式の g を F でずりっと下ろしてやると——

$$\begin{array}{ccccc} & & Y & \xrightarrow{g} & F(Z) \\ X & \xrightarrow[f]{} & F(Y) & \xrightarrow[F(g)]{} & F^2(Z) \end{array}$$

となって、あー……これを合成しても $X \longrightarrow F^2(Z)$ にしかならないけど関手 F がモナドならば自然変換 μ を使ってこんな風にできる——

$$\begin{array}{ccccc} & & Y & \xrightarrow{g} & F(Z) \\ & \nearrow & & \circlearrowleft & \uparrow \mu_Z \\ X & \xrightarrow[f]{} & F(Y) & \xrightarrow[F(g)]{} & F^2(Z) \end{array}$$

[*2] Heinrich Kleisli (October 19, 1930 April 5, 2011)

第3章

【新編】加速しなイカ？【叛逆の物語】

— @nushio

希望を願い、呪いを受け止め戦い続ける者たちがいる。それが科学少女。
就職を拒んだ代償として戦いの運命を重ねた魂。その末路は、消滅による救済。
この世界から消え去ることで、絶望の因果から解脱する。
いつか訪れる終末の日、“技術的特異点”を待ちながら、私たちは戦い続ける。
剽窃と捏造ばかりを繰り返す、この救いようのない世界で。
あの、懐かしい笑顔と、再びめぐり合うことを夢見て。

3.1 プロローグ

見滝原中学校　教室

「…女子のみなさんは、くれぐれもジャバじゃなきゃアイティーじゃない、とか抜かす男とは交際しないように！」

「そして男子の皆さんは、絶対に特定の言語にケチをつけるような大人にならないこと！」

生徒達の苦笑。早乙女先生の失恋話はもはや生徒たちにとって慣れっこらしい。私にとっても——そう、数えるのを諦めるほどに。黒板には意味をなさないコードの断片が手描きフォントで貼り付けられている。シャフトちゃんと仕事しろ。

「コホン♡それでは、今日の授業は先週の宿題から。好きなプログラミング言語を選んでその歴史についての調べ学習でしたね。暁美ほむらさん」

「はい」

ここで当てられるのは中沢くんではなく私。それは運命石（シュタインズゲート）の選択——ではなく、私が意図的に引き当てた世界線なのだ。英語の授業がおかしな言語の授業へ置き換わっていることもそう。物語を操るモナドの力に辿り着くために。物語の外に逝ってしまった彼女にもう一度めぐり逢うために。私は落ち着いたペースで教室前方へ歩を進める。ノート PC などは携えず手ぶら。地域モデル校先行導入の教育支援情報クラウドシステムにより、電子黒板が自動的に私のデスクトップに切り替わる。早乙女先生と場所を替わった私はプレゼンテーションを始めた。

3.2 夢のようなライブラリがあった、ような…

「かつて、CPU の周波数が時間の指数関数的に向上を続け、ただ互換性のある CPU を買い換えつづけるだけで、あらゆるソフトウェアの性能が向上していったフリーランチとよばれる時代がありました。しかし、西暦 2005 年ごろには CPU の単一コアの周波数の上昇傾向が終わり、以降は誰もがいつまで待っても到着しないランチを狩りに行くために並列プログラミングという重労働を強いられる時代となりました。CPU も Bulldozer や Haswell など、どんどんベクトル化／マルチコア化

第4章

ご注文は依存型ですか？

— @tanakh

こころぴょんぴょん待ち？
考えるふりして
もうちょっと！ 近づいちゃえ。

```
main :: IO ()
main = mapM_ (putStrLn . ah) [1..105]

ah :: Int -> String
ah n = fromMaybe (show n) $ f <> g <> h where
  f = if n ‘mod‘ 3 == 0 then Just "あぁ＾～～、" else Nothing
  g = if n ‘mod‘ 5 == 0 then Just "こころが" else Nothing
  h = if n ‘mod‘ 7 == 0 then Just "ぴょんぴょんするんじゃ～" else Nothing
```

簡単には教えないっ！
こんなに好きなことは～～～～～（テンテテン！）
内緒なのぉ～～～～！

```
1
2
あぁ＾～～、
4
こころが
あぁ＾～～、
ぴょんぴょんするんじゃ～
8
あぁ＾～～、
こころが
11
あぁ＾～～、
13
ぴょんぴょんするんじゃ～
あぁ＾～～、こころが
16
```

```
17
あぁ^〜〜、
19
こころが
あぁ^〜〜、ぴょんぴょんするんじゃ〜
22
23
あぁ^〜〜、
こころが
26
あぁ^〜〜、
ぴょんぴょんするんじゃ〜
29
あぁ^〜〜、こころが
31
32
あぁ^〜〜、
34
こころがぴょんぴょんするんじゃ〜
あぁ^〜〜、
37
38
あぁ^〜〜、
こころが
41
あぁ^〜〜、ぴょんぴょんするんじゃ〜
43
44
あぁ^〜〜、こころが
46
47
あぁ^〜〜、
ぴょんぴょんするんじゃ〜
こころが
あぁ^〜〜、
52
53
あぁ^〜〜、
こころが
ぴょんぴょんするんじゃ〜
あぁ^〜〜、
58
59
あぁ^〜〜、こころが
```