Haskell コンパイラを書こう！

書いた日： 2016 年 12 月 15 日、書いた人：@unnohideyuki[*1]

はじめに

これは、Haskell Advent Calendar 2016[*2] 15 日目の記事です。

この記事では、私が Haskell コンパイラを Haskell で書いてみたいと思って勉強した事柄を紹介したいと思います。Haskell は多くの特徴をもった高水準言語であるため、Haskell コンパイラが行わなければならない仕事はたくさんあります。しかし、それらを一か所にまとめて述べた教科書のようなものは見当たらなかったので、結構たくさんの文書を調べてまわる必要がありました。ここで、それら全てについて詳しく解説するには、紙幅も私の能力も足りませんが[*3]、簡単な紹介と参考文献へのリンクを示しておけば、多少の価値はあるのではないかな…、あればいいな、と期待しています。

最初のターゲット: qsort

コンパイラに限らず、そこそこの規模のプログラムを作ろうとする場合には、最初のマイルストーンとなるようなサブ目標を決めるのが良いように思われます。有名な Wnn における「私の名前は中村です」のようなやつ。しかし、ここで Hello, world では Haskell らしさがあまりでないような気がするので、もう少しそれっぽいものにしたいです。そこで私は、当時のお気に入りで、かつ、ちょっと不思議すぎて嫌いでもあった qsort を最初の目標に定めることにしました。

以下に示すのが、その「最初の目標」プログラムです。型宣言をコメントにしてあるのは、これらはコンパイラに推論させた方が面白かろうと当時考えたためです。

```
-- qsort :: Ord a => [a] -> [a]
```

[*1] https://twitter.com/unnohideyuki

[*2] http://qiita.com/advent-calendar/2016/haskell

[*3] いつかやってみたいですが。

```
qsort []     = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
               where
                 smaller = [a | a <- xs, a <= x]
                 larger  = [b | b <- xs, b > x]

-- main :: IO ()
main =
  do let helo = "Hello, World!"
     putStrLn helo
     putStrLn.show $ qsort [3, 1, 4, 1, 5, 9, 2, 6, 5]
     putStrLn $ show $ qsort helo
```

いま思うと、これを最初の目標に定めたのは、我ながら絶妙だったなという気がします。型クラスあり、パラメトリック多相もあり、パターンマッチもあるし、リスト内包式も登場します。do 記法もありますね。実は、私の作ってるコンパイラがこのプログラムを扱えるようになったのが、つい数日前だったという意味でも絶妙でした。

で、ひとまずこれを扱えるコンパイラを書くぜ！と決めた私が、次にやろうとしたことは、「GHC のソースを読もう！」だったのですが……、これは無謀でした。Haskell が得意でもなく、コンパイラを書いたこともない人間が、GHC のソースなんて読めっこありません。だめだー。だめだった。

というわけで、お、そういえば Tiger book 持ってたわ、ということに気付きました。

ターゲットマシン: Android (Java)

もうひとつ、コンパイラを作る上で重要な選択に、ターゲットマシンを何にするかというのがあります。私は、ターゲットマシンは Android にしました。これは、個人的な理由なのですが、私は Haskell プログラミングが出来るようになりたいと思っていると同時に、Android アプリも書いてみたいとも思っていたので、Haskell で Android アプリが書けるようになればイイじゃないか、というわけでして。

コンパイラの基礎: Tiger book を読もう

Modern compiler implementation in ML という書籍があります（通称 Tiger book, 邦訳は「最新コンパイラ構成技法」[*4]）。この本は、非常に有名なコンパイラの教科書で、Tiger という名の小さなプログラミング言語の処理系を少しずつ作っていく過程で、コンパイラ作成に必要な技術を学んでいけるようになっています。この本は、とても良い本だとは思うのですが、独学で読み通すのは結構むずかしく、さらに、ML にあまり親しみのない身にはつらいところもあります。私は、この本を買ったはいいけど、読み進められずに頓挫していました。

そこで、Tiger book へのリベンジを兼ねて、この本を ML ではなく Haskell で Tiger コンパイラを書きな

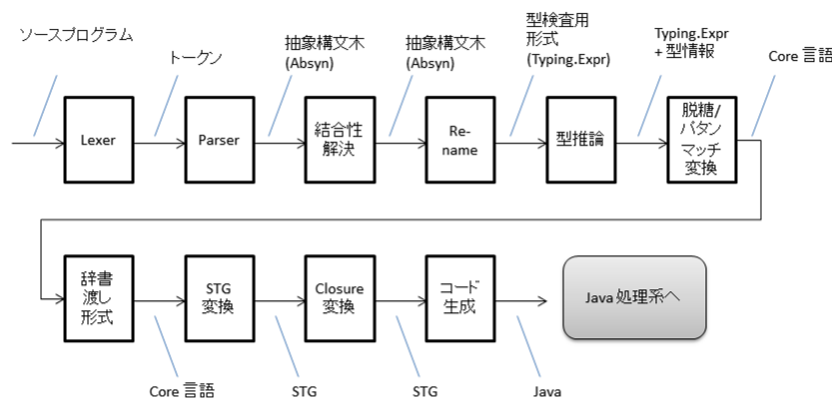
[*4] 私は最初（まだ邦訳がでる前）英語版を読み始めて、結局翻訳版も手に入れたので、両方持ってます。

がら読み進めることにしました。幸い、ML と Haskell は、この本を読み進めるうえでは支障がほとんどない程度には、似ています。ML で ref 型をつかって代入していたところを、State モナドを使って書き換える必要があったくらいでしょうか。「Haskell できるようになりたい」という熱意も加わって、どうにか、この本の基本編（前半）を踏破することができました[*5]。

そういうわけで、Haskell で Haskell コンパイラを書く練習として、まず、Haskell で Tiger コンパイラを書いてみたわけですが、Haskell 初心者でコンパイラを書いた経験もなかった私にとっては、Haskell でなにかまとまった分量のプログラムを書いてみるという経験と、とにかくコンパイラを書いたことがある状態への遷移という一挙両得で、お得でした。急がばまわれ、じゃなくて、そこに教科書があるなら、それを読むのが最短コースというわけですね、きっと。

Haskell コンパイラの処理フェーズ

わたしが今回つくった Haskell コンパイラの処理フェーズを下図に示します。これは、他のコンパイラ実装では多少順番が異なったりするかもしれません。



構文解析 (Syntactic Analysis = Lexer + Parser)

構文解析には、Alex と Happy を用いました。これは、古くからある Lex + Yacc の Haskell 版で、それらを素直に Haskell に移植したような形をしていますが、各言語に数多ある Lex + Yacc クローンのなかでも、かなり良い出来栄えのツールなのではないかと思います。特に、Semantic Action を Haskell プログラムとして書くおかげで、型検査が効くところが素晴らしい。

構文解析フェーズにおいて、Haskell コンパイラ特有の処理としては、以下の２つを挙げることができます。

- レイアウト規則
- 結合性の解決 (Fixity Resolution)

レイアウト規則の仕様は、Haskell 2010 Language Report の 10.3 節[*6]にて規定されています。また、これを実装するためには、Lexer と Parser が共通の「状態」を持つ必要があるのですが、そのためには

[*5] 聞くところによると、ムツカシイ本を読破する秘訣は、半分読んだら「読破した」ことにすることだそうです。まったくだと思います。

[*6] https://www.haskell.org/onlinereport/haskell2010/haskellch10.html#x17-17800010.3

monadUserState wrapper[*7] という仕組みが使えます。Alex + Happy の先輩である Lex + Yacc にも、Lexer と Parser が状態を共有する仕組みはあって、要するにグローバル変数だったりしたのですが、Haskell でやるなら状態モナドですよね。

もうひとつ、結合性の解決 (fixity resolution) については、Haskell 2010 Language Report の 10.6 節[*8] で規定されています。Haskell では、中置演算子をプログラム中で自由に定義することができて、結合性（右結合なのか、左結合なのか、はたまた結合しないのかとか、結合の強さも）までもがプログラム中で指定できるようになっています。そのため、構文解析では、いったん結合性不明のまま構文解析しておいて、その後段で、指定された結合性に基づいて構文木を変換します。ちなみに、結合性の宣言は、その中置演算子が初めて出現する場所より後ろにあってもいいため、構文解析と fixity resolution が「２パス」になるのは避けられません[*9]。

意味解析 (Semantic Analysis)

構文解析は、入力された文字の並びを、Haskell 文法に従うものと、そうでないものに峻別し、前者については構文木を作成して、後者については構文エラーを報告します。言い換えると、Haskell の文法に基づいて、Haskell プログラムとそれ以外を分類して、Haskell プログラムといえるものだけを後続のフェーズに渡すわけです。

その次の段階が「意味解析」なのですが、ここでは何が行われるべきでしょうか。前にならうと、次のように言えそうです：構文解析器がよこしてきた、文法的には正しい Haskell プログラム様のものを、Haskell プログラムとして意味を持つものと、持たないものに峻別する。そして、意味をもつものに対しては、適切な意味情報を与え、そうでないものについてはエラーを報告する。

たとえば、次のような「プログラム」は、文法的には Haskell プログラムのように見えますが、ちゃんとした意味を持たないため、実行可能なプログラムとして解釈することが出来ません。

```
x = 1
y = "two"

main = putStrLn $ x + y
```

この意味解析で中心的な役割をするのが「型」です。Haskell のような静的型付け言語では、ソースプログラムの各項について型検査が行われ、型が不一致するものや型が曖昧で定まらない（つまりセマンティクスが定まらない）ものについては、この段階でエラーとなります。そのため、この類のエラーが実行時に発覚することがありません[*10]。

また、Haskell では[*11]、プログラマが全ての変数の型を宣言する必要はなく、型検査器が「型推論」してくれます。このような型推論を特徴とする型システムとしては、Hindley/Milner の型システムが有名ですが、

[*7] https://www.haskell.org/alex/doc/html/wrappers.html#id462498
[*8] https://www.haskell.org/onlinereport/haskell2010/haskellch10.html#x17-18100010.6
[*9] monadic parser で上手いことやれないかなと思ったんですが、ダメでした。
[*10] しばしば「型安全」といわれる。
[*11] ML のような先輩言語においても、そうですが。

Haskell の型システムは、Hindley/Milner 型システムに拡張を施したものとなっています。

Haskell の型システムが、Hindley/Milner 型システムに対して、どういった動機で、どういう拡張を施したものなのかについては、How to make ad-hoc polymorphism less ad hoc[*12] という論文に詳しく述べられています。簡単にいうと、パラメトリック多相が Hindley/Milner 型システムによって、非常にうまく扱われてきたのに対し、多重定義の扱いがいまいちバラバラなので、なんとかしたいよね。型クラスを導入すれば、多重定義も型システムに組み込むことができ、統一的に扱えますよといったことが書かれています。

なお、Haskell の型推論・型検査器については、Typing Haskell in Haskell[*13] という、とても有り難い論文があって、そこでは、Haskell の型検査器を実際にうごく Haskell プログラムとして示してあります。こんな有り難いものを用いない手はありませんので、私が Haskell コンパイラを作る際には、Typing Haskell in Haskell のコードを使わせていただきました。

ちなみに、Haskell も英語も得意でない私にとって、この Typing Haskell in Haskell を読むのも楽ではありませんでした。そこで、ほとんど日本語に書き写すようにして二回ほど読んだのですが、そのノートも公開しておきますので、誰かがこの論文を読む助けになれば幸いです。

リネーミング、脱糖、パターンマッチの変換

意味解析フェーズでは、型推論・型検査を行ったうえで、構文解析木を Core 言語と呼ばれる、中間言語に変換します。Core 言語への変換に際しては、型推論以外にもいくつかの処理が必要となります：

- 変数のリネーミング（α 変換）
- 脱糖 (desugar)
- パターンマッチの変換

まず、すべての変数は α 変換によって、ユニークな名前を持つようにリネームされます。これは、型推論より前に実施します。

つぎの脱糖とは、 Haskell にあって Core 言語にない文法の要素を、変換して、Core 言語で表現可能な形になおすことです。これは、興味深いのですが、後述するパターンマッチの変換を除けば、そんなに難しくはありません。たとえば、リスト内包式をどのように変換したらいいかについては、3.11 節[*14] に述べられており、do 記法については 3.14 節[*15] にあるというように、Haskell 2010 Language Report を見れば、脱糖の方法が書かれています。

で、結構むずかしい（と私は思った）のが、パターンマッチの扱いです。Core 言語には、単純な case 式しかないため、パターンマッチはすべて「平坦な」case 式に書き換えてやる必要があります。たとえば、

```
mappairs f [] _ = []
mappairs f _ [] = []
mappairs f (x:xs) (y:ys) = f x y : mappairs f xs ys
```

[*12] http://dl.acm.org/citation.cfm?id=75283
[*13] https://web.cecs.pdx.edu/~mpj/thih/
[*14] https://www.haskell.org/onlinereport/haskell2010/haskellch3.html#x8-420003.11
[*15] https://www.haskell.org/onlinereport/haskell2010/haskellch3.html#x8-470003.14

は、

```
mappairs = \f xs' ys' ->
  case xs' of
    [] -> []
    (x:xs) -> case ys' of
                [] -> []
                (y:ys) -> f x y : mappairs f xs ys
```

のように変換されなければなりません。

このパターンマッチについては、The Implementation of Functional Programming Languages[*16] の 5 章、Efficient Compilation of Pattern-Matching が非常にわかりやすくて助かりました。ここでは、Miranda による実装例も示されており、例によって Miranda と Haskell も十分似ているため、もってきて、動かしてみて確かめることが出来ました。

Core 言語

Core 言語は、GHC が採用している型付きの中間言語です。Core 言語に言及した文書はいくつかあるのですが、Core 言語そのものにフォーカスしたものとして、Type directed compilation in the wild GHC and System FC[*17] を参照するのが良いと思います。ここでは、型付きの中間言語を持つメリットとして、以下の３点が挙げられています。このおかげで、GHC は積極的な拡張を続けながらも正気を保つ (”stay sane”) ことが出来ているんだとか。

1. 中間言語が小さければ、分析、最適化、および、コード生成はいずれも、小さな言語を扱えばよくなる
2. Core に対する型検査は、コンパイラの内部処理に対する強力な一貫性チェックになる
3. Core の設計は、ソース言語における病的な型システム拡張に対する、強力な正常性確認となる。もし、Desugar してうまく Core に変換できるなら良いが、無理ならその拡張は再考した方がいい

オリジナルの Core 言語の定義は、以下の通り、たった 3 つの型と、15 のコンストラクタから成ります：

```
data Expr
  = Var Var
  | Lit Literal
  | App Expr Expr
  | Lam Var Expr -- Both term and type lambda
  | Let Bind Expr
  | Case Expr Var Type [(AltCon, [Var], Expr)]
```

*16 http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/
*17 https://www.cs.uoregon.edu/research/summerschool/summer13/lectures/FC_in_GHC_July13.pdf

```
  | Type Type -- Used for type application

data Var = Id Name Type -- Term variable
  | TyVar Name Kind -- Type variable

data Type = TyVarTy Var
  | LitTy TyLit
  | AppTy Type Type
  | TyConApp TyCon [Type]
  | FunTy Type Type
  | ForAllTy Var Type
```

今回私がつくったコンパイラでは、型の表現としては Typing Haskell in Haskell のものを使ったので、これとは異なっているのですが、表現される内容は同じです (ForAllTy は (Qual Type) であらわされるなど[*18]、表現の仕方はことなりますが。)。

辞書渡しスタイルへの変換

Core 言語は、このあと、さらに単純な中間表現である STG に変換されるのですが、Core 言語において明示されていた型情報は、STG に変換されるときに取り除かれてしまいます。なので、多相を扱うために必要な変換は、型情報を取り除く前に、つまり Core 言語に対して行う必要があります。

ひとつは、パラメトリック多相の扱いですが、これについては、大きくわけて以下のような実現方法があります。ここでは、単純な fully-boxing でいくことにしました。まずは、性能を気にしなければ、単純につくれそうなことと、ターゲットを Java 実行系（Android なんですけど）にすることにしたので、オブジェクトが boxed になるのは、まぁ、仕方がなさそうでもあるので。

- fully-boxing : 全てのオブジェクトを参照型として扱うので、関数に渡すときには常に同じサイズ（ポインタのサイズ）
- type-passing : 型情報を引数として渡す、オブジェクトのサイズは型ごとに異なってよい
- コード複製・展開 : 各型ごとに対応する関数を必要なだけ生成して用いる

Fully-boxing にすれば、多相は全部あつかえるかというと、そうではありません。引数を全て参照型にして、その型に関するメソッドテーブルもそこから辿れるようにしたとしても、型クラスに関する多相を解決できないことがあります。たとえば、fromIntegral のように、戻り値の型に応じて適切な処理を選ばなければならないケースです。

```
Prelude> :t fromIntegral
fromIntegral :: (Integral a, Num b) => a -> b
```

[*18] 誤記訂正 2020-04-23

この実現のため、型クラスによって多重定義される関数は、メソッドの辞書を受け取って、それによって適切なメソッドを呼び出せるようにします。多重定義される関数の仮引数に、辞書をうけとるための引数を追加したり、そういった関数に辞書を渡すような判断には、型情報が必要となるため、Core 言語を STG に変換する前に実施します。

私は、型推論の後段に DictPass というモジュールをつくって、そこにこのための処理を書きました。これは、Typing Haskell in Haskell の型検査器をなるべく変更せずにそのまま用いたかったからですが、辞書渡し形式に変換するために必要な計算は、型推論と重複するところも多いので、本当は、これらのモジュールは、一体化してもっときれいに書きなおすのがいいように思います。

辞書渡しについては、先ほど紹介した How to make ad-hoc polymorphism less ad hoc で述べられているのですが、それに加えて、Dictionary-free Overloading by Partial Evaluation[*19] という論文の前半を読むとわかりやすくて良かったように思います。

STG

Core 言語は、さらに STG と呼ばれる中間言語に変換されます。この STG の特徴はとしては、Core と十分似ているので都合が良いこと、さらに、非正格の高階言語としてのセマンティクスが定められていることが挙げられます。

STG については、Making a Fast Curry: Push/Enter vs. Eval/Apply for Higer-order Languages[*20] に書かれています。ここでは、Push/Enter と Eval/Apply の二種類の評価方式が述べられているのですが、この論文でより優れているとされている（よって GHC も採用している）Eval/Apply rules を用いることにしました。

STG のもうひとつの特筆すべき特徴としては、STG を変換した結果としてできる実行系では、全ての関数呼び出しは末尾呼び出しにできる、というものがあります。ですが、今回ターゲットマシンに選んだ Java 処理系では、末尾呼び出しはできないこともあり、また、まずは動くものを作りたかったこともあって、今回つくったコンパイラ向けには、evaluation rules を素直に実行するようなランタイムを用意しました[*21]。

クロージャ変換

コード生成の前にやっておかなければならない仕事として、クロージャ変換があります。クロージャ変換とは、自由変数を持つような関数があったときに、その自由変数を、その関数のすぐ外がわで束縛してやることで、自由変数を無くすような変換のことです。これは、Haskell のように関数がファーストクラスで、かつ、レキシカルスコープを持つような言語のコンパイラには必須の処理です。

たとえば、関数 f とその自由変数を x1, ..., xn としたとき、f のクロージャ変換とは、f を次のような形に変換することを指します。

[*19] http://web.cecs.pdx.edu/~mpj/pubs/pepm94.pdf
[*20] http://community.haskell.org/~simonmar/papers/evalapplyjfp06.pdf
[*21] なので、STG の優れた点は、今回のコンパイラではほとんど生かされていないなという感はあります。

```
(\x1 ... xn -> f) x1 ... xn
```

これは、let 式を用いると次のようにも書けます。

```
let
  x1’ = x1
  x2’ = x2
    ...
  xn’ = xn
in
  f’  -- (f’ は f における x1, ..., xn を x1’, .., xn’ に置き換えたもの)
```

こうして変換しておくことで、関数は、いつどこから呼ばれても、正しく動作することが可能となります。

ところで、Tiger Book の前半をこなしただけでは、このあたりをどう処理していいのかわからず、随分悩みました。ところが、「クロージャ変換」という、必要な処理の名前がわかってしまえば、参考文献はたくさん見つかります。Tiger Book の 15.5 節にも書いてありました。

コード生成

Tiger コンパイラを作ったときには、出力コードは Dalvik のアセンブリ言語にしたのですが、今回は、Java を吐くようにしました。これは、結果的には大正解だったと思います。Tiger のコード生成は、すでに十分低レベルな TREE 言語を変換すればよかったので、Dalvik に変換するのも、それほど苦にはならなかったのですが、STG は、それに比べるとずっと複雑なので、Dalvik を吐くことにしていたら、まだ出来てなかったような気がします。また、Java を吐くことにしたので、普通の Java プログラムとして、PC 上で実行できるのも良かった点です。

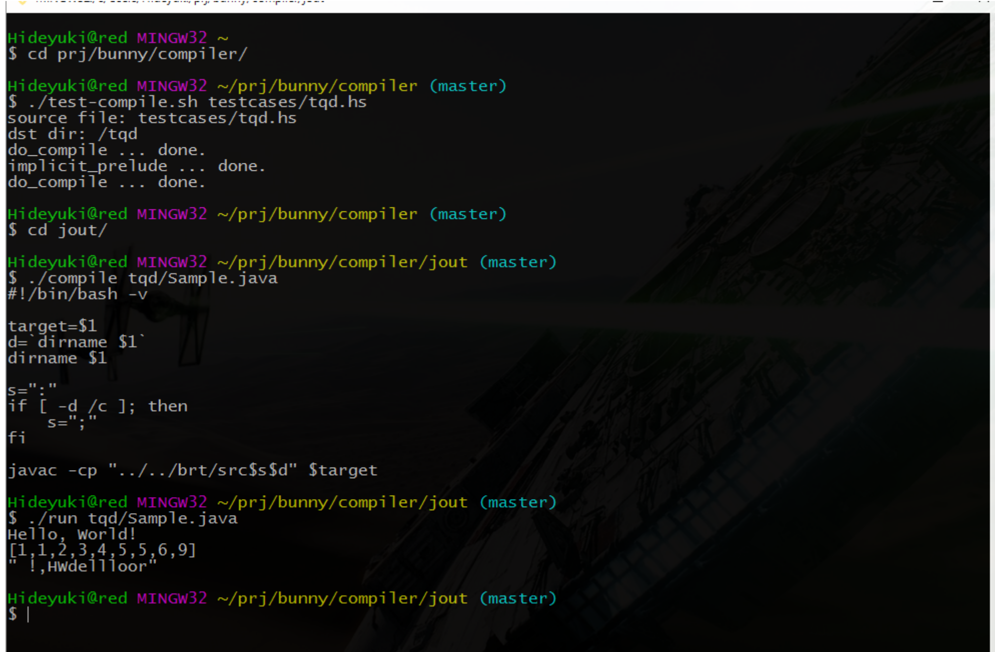
そのコード生成については、ほとんど自己流なので、ここでご紹介するような事柄は、特にないように思います。辞書渡しで受け渡される辞書は、Java におけるメソッドテーブルであるところのクラスで表現しました、とか、くらいでしょうか。

ただ、STG から直接 Java プログラムを文字列としてコード生成している点については、最初の実装としては良かったと思っていますが、近い将来直したいと思っています。できれば、GHC における C– のように、Java のサブセットを表現するような中間言語 (J–) に変換しておいてから、最後に単純変換で Java プログラムを吐くようにしたいです。そうすれば、見通しが良くなるだけでなく、この中間言語に対して多少の最適化のチャンスも生まれそうですし、もしかすると、Java/Objectiv-C の二種類のターゲットをサポートできたりも、するかもしれません。

それらをひとまとめに

ここまで、Haskell コンパイラを書くために調べたことを紹介してきました。そうやって出来たコンパイラのソースコードは、GitHub 上で公開*22 してあります。

…とはいっても、まだ出来上がってはいなくて、冒頭にあげた qsort プログラムをやっとコンパイル、実行できるようになったばかりです。しかも、ターゲットマシンを Android にするといいつつ、まだ Android 向けのランタイムを用意できておらず、普通に PC 上での実行しかできません。



そういうわけで、まだ、「使ってみようかな」という向きにはおすすめできませんし、あやふやな理解のまま書いてしまった部分も多く残っているため、コードも汚いです。ですので、今後は、だいたい以下のようなことをやって「バージョン 1」を目指します：

- 未実装機能をなくしていって、Haskell 2010 の範囲をサポートする
- 復習を兼ねたコードレビューとリファクタリング（現状きちゃなすぎ）
- Android 向けランタイムとコンパイル環境の整備
- Foreing Function Call で Java のライブラリを呼べるようにする

性能を問題にするのは、その後、ですよね。

まとめ

Haskell で Haskell コンパイラをつくろうということで、私自身が Haskell コンパイラを書くために調べたことがらを紹介させていただきました。まだ、コンパイラの実装は未完成で、「みなさん、使ってみてください」と言えないのが残念ではありますが、近い将来ご紹介できるようにしたいと思います。

---

[22] https://github.com/unnohideyuki/bunny

ところで、私は、今回のコンパイル目標にした qsort のようなプログラムが Haskell で書けることについては、すごいな、素敵だなと思う一方で、不思議すぎて、なんだか嫌だったんですよね。この嫌な感じを克服するのと、たくさん Haskell コードを書いて慣れてしまうことの両方をめざして、Haskell 初心者だったにも関わらず、Haskell コンパイラの作成に挑んでみたのでした。結構大変なので、目的に対して割にあっているのかどうか全然わかりませんが、個人的にはすごく面白かったので、よかったかなと思います。

それでは、この辺で、この記事はおしまいです。