Haskellを書きたい人生だった

C++しょしんしゃかい くそざこはっぴょう

phi16

Haskellが好きです

(Haskellはただのプログラミング言語です)

自己紹介

- phi16
- Twitter:@phi16_
- Haskellでbot組んだり
- Haskellでゲームつくったり
- Haskellでレイトレ組んだり(コンパイル時じゃないです)
- HaskellでWebサーバ組んだり
- Haskellで3Dゲームつくります(予定)

- TitechB1

Note

- ゆるふわにいきます

- C++コードは
  - 型がオレンジ
  - で色つけてるので参考に
  - 色々略すので詳しく知りたければ実コードを

- 細かい部分話さないのであとで見てください
  - アップロードするので

経緯

- 面白そうな勉強会が
- 自由参加枠締め切り
- 発表枠空いてる！
- 変なアイディア思いついた！
- いこう！

経緯

- 面白そうな勉強会が
- 自由参加枠締め切り
- 発表枠空いてる！
- 変なアイディア思いついた！
- いこう！

なんのはなし

- C++書くのはつらい
- Haskell書きたい！
- Haskell書こう！
- でもC++の勉強会らしい
- C++でHaskell書けばいいのでは？？？

結論(Haskell)

```haskell
main = getLine >>= ¥x ->
  (putStrLn $ if  x == “C++”
              then  “No more C++”
              else  “Hello, “ ++ x) >>
  (print $ let f x = 2 * x in f 5) >>
  (print $ take 10 $ map (¥x -> 2*x) [1..]) >>
  (print $ filter (¥x -> x /= 0) [1,0,8,0,0]) >>
  return ()
```

結論(Haskell)

```
main = getLine >>= ¥x ->
  (putStrLn $ if  x == “C++”
                then  “No more C++”
   関数適用↓      else  “Hello, “ ++ x) >>
  (print $ let f x = 2 * x in f 5) >>  ↓無限リスト
  (print $ take 10 $ map (¥x -> 2*x) [1..]) >>
  (print $ filter (¥x -> x /= 0) [1,0,8,0,0]) >>
  return ()                          ↑否定比較演算子
```

結論(Haskell)

```haskell
main = getLine >>= ¥x ->
  (putStrLn $ if  x == “C++”
              then  “No more C++”
              else  “Hello, “ ++ x) >>
  (print $ let f x = 2 * x in f 5) >>
  (print $ take 10 $ map (¥x -> 2*x) [1..]) >>
  (print $ filter (¥x -> x /= 0) [1,0,8,0,0]) >>
  return ()
```

結論(C++)

```
main_= getLine >>= x -->
 _(putStrLn $ if_ x == “C++”
                then  “No more C++”
                else_ “Hello, “_+++x) >>
 _(print $ let f x = 2 * x in f 5) >>
 _(print $ take 10 $ map_(x --> 2*x) [1__]) >>
 _(print $ filter_(x --> x /= 0) [1_,0,8,0,0]) >>
  return_()
```

結論(C++)

```
#include”cpp.hs”

main_= getLine >>= x -->
 _(putStrLn $ if_ x == “C++”
              then  “No more C++”
              else_ “Hello, “_+++x) >>
 _(print $ let f x = 2 * x in f 5) >>
 _(print $ take 10 $ map_(x --> 2*x) [1__]) >>
 _(print $ filter_(x --> x /= 0) [1_,0,8,0,0]) >>
  return_()

-- End Haskell
```

結論(C++)

```
phi16@miyu~/Source/C++/Haskell% cat main.cpp
#include"cpp.hs"

main_= getLine >>= x -->
 _(putStrLn $ if_ x == "C++"
                then  "No more C++"
                else_ "Hello, "_+++ x) >>
 _(print $ let f x = 2 * x in f 5) >>
 _(print $ take 10 $ map_(x --> 2*x) [1__]) >>
 _(print $ filter_(x --> x /= 0) [1_,0,8,0,0]) >>
   return_()

-- End Haskell
```

結論(C++)

```
phi16@miyu~/Source/C++/Haskell% time clang++ -std=c++1y main.cpp
clang++ -std=c++1y main.cpp  5.24s user 0.21s system 99% cpu 5.468 total
phi16@miyu~/Source/C++/Haskell% ./a.out
C++
No more C++
10
[2,4,6,8,10,12,14,16,18,20]
[1,8]
phi16@miyu~/Source/C++/Haskell% ./a.out
Haskell
Hello, Haskell
10
[2,4,6,8,10,12,14,16,18,20]
[1,8]
```

結論(C++)

```
phi16@miyu~/Source/C++/Haskell% time clang++ -std=c++1y main.cpp
clang++ -std=c++1y main.cpp  5.24s user 0.21s system 99% cpu 5.468 total
phi16@miyu~/Source/C++/Haskell% ./a.out
C++
No more C++
10
[2,4,6,8,10,12,14,16,18,20]
[1,8]
phi16@miyu~/Source/C++/Haskell% ./a.out
Haskell
Hello, Haskell
10
[2,4,6,8,10,12,14,16,18,20]
[1,8]
```

あじぇんだ

- 技術解説
  - template
  - E.T.
- 実装
  - 作った処理を順番に紹介
- 余談

template

- 普通の関数

```
int f(int t){
  return t + t;
}
```

```
> f(4) ≡ 4 + 4 ≡ 8
```

template

- 普通の関数

```
std::string f(std::string t){
  return t + t;
}

std::string str = “abc”;
> f(str) ≡ str + str ≡ “abcabc”
```

template

- 普通の関数

```
T f(T t){
  return t + t;
}
```

template

- 複数の型に対しておなじ処理を書くときに同一化できる

```
template <class T> T f(T t){
  return t + t;
}
```

```
> f(4) ≡ 4 + 4 ≡ 8
std::string str = “abc”;
> f(str) ≡ str + str ≡ “abcabc”
```

template

- 構造体にも使える(Container等によく使われる)

```
template <class T, class U> struct Pair{
  T t; U u;
  Pair(T t, U u):t(t),u(u){}
};

Pair<int,char> pos(5,'a');
> pos.t == 5 && pos.u == 'a'
```

template

- 型の一部が違うときでも使える

```
template <class T> std::size_t len(std::vector<T> t){
  return t.size();
}

std::vector<int> a{1,2,3,4};
std::vector<std::string> b(100);
> len(a) == 4 && len(b) == 100
```

template

- 型が違っても使える

```
template <class T> struct A{};
template <> struct A<int>{static const int a = 1;};
template <> struct A<char>{static const int a = 2;};

> A<int>::a == 1 && A<char>::a == 2
```

- 関数のときは普通にオーバーロード

型で計算

```
struct Zero{};
template <class T> struct Succ{T t;};

int value(Zero z){
  return 0;
}
template <class T> int value(Succ<T> s){
  return 1 + value(s.t);
}
```

型で計算

```
> value(Succ<Succ<Succ<Zero>>>()) == 3
```

- templateで型レベルだけで演算できる
  - チューリング完全なのでなんでも組めます
    - 再帰制限なければ

話は変わって

operator

```
struct Int{
  int v;
  Int(int v):v(v){}
  Int operator +(Int a){
    return Int(v + a.v);
  }
};

> Int(3) + Int(6) == Int(9)
```

operator

```
Int operator +(Int a, int b){
  return a + Int(b);
}
Int operator +(int a, Int b){
  return Int(a) + b;
}

> 3 + Int(6) == Int(9)
> Int(1) + 2 + 3 + 4 == Int(10) //左結合
```

operator

- 誰も足し算しなきゃいけないとは言ってない

```
struct Log{
  int v;
  Log(int v):v(v){}
  Log operator +(Log a){
    return Log(v * a.v);
  }
};

> Log(2) + Log(3) == Log(6)
```

operator

- 誰も自分を返さなきゃいけないとは言ってない

```
template <class T> std::size_t operator -(std::vector<T> v){
  return v.size();
}

std::vector<int> v{1,2,3,4,5};
> -v == 5 //みためやばい
```

operator

- templateで独自クラスを返す

```
Pair<Int,Int> operator +(Int a, Int b){
  return Pair<Int,Int>(a,b);
}

> (Int(3) + Int(4)).u == Int(4)
// Int(3) + Int(4) の型は Pair<Int,Int>
```

operator

- ついでにもっと

```
template <class T, class U, class V, class W>
          auto operator +(Pair<T,U> t, Pair<V,W> u){
  return Pair<Pair<T,U>,Pair<V,W>>(t,u);
}
```

(Int(1)+Int(2)) + (Int(3)+Int(4)) の型は
Pair<Pair<Int,Int>,Pair<Int,Int>>

そんなノリ

```
x の型が X
x + y の型が Add<X,Y>
x * y + z の型が Add<Mul<X,Y>,Z>
(x + y) ^ 2 - 2 * x * y の型が
    Sub<Pow<Add<X,Y>,Int>,Mul<Mul<Int,X>,Y>>
```

- Expression Template
  - 式の構造をtemplateで保持
  - Vector計算の一時ObjectのEliminationに使ったりする(らしい)

他の話は即時解説します

実装

```
main = print $ take 10 $ map (¥x -> 2*x) [1..]
```

実装

```
#include”cpp.hs”

main_= print $ take 10 $ map_(x --> 2*x) [1__]

-- End Haskell
```

マクロ展開

```
/* 前略 */

int main(){hs[_=

  &print_& &apply& &take_& 10 &apply& map_(x --> 2*x) [1__]

-- ]; return 0;}
```

整形

```
/* 前略 */

int main(){
 hs[
  _= &print_& &apply& &take_& 10 &apply& map_(x --> 2*x) [1__] -
 ];
 return 0;
}
```

整形

```
int main(){
 hs[
  _= &print_& &apply& &take_& 10 &apply& map_(x --> 2*x) [1__] --
 ];
 return 0;
}
```

整形

```
hs[
 _= &print_& &apply& &take_& 10 &apply& map_(x --> 2*x) [1__] --
];
```

外枠

```
hs[
 _= <Expr> --
];
```

- hs[] 内部がHaskell式として評価される
  - _へのoperator=は右辺をそのまま返す
    - 元はmain_への代入
  - 最後につくoperator--は左辺をそのまま返す
    - 元は -- End Haskell ←Haskellのコメント
- Haskellっぽくmainを書くための部分

外枠

```
hs[
 _= &print_& &apply& &take_& 10 &apply& map_(x --> 2*x) [1__] --
];
```

外枠

```
hs[
    &print_& &apply& &take_& 10 &apply& map_(x --> 2*x) [1__]
];
```

- hs[] は評価関数呼んでるだけ

式

```
&print_& &apply& &take_& 10 &apply& map_(x --> 2*x) [1__]
```

式

```
&print_& &apply& &take_& 10 &apply& map_(x --> 2*x) [1__]
```

- 各Objectの先頭と最後に&を入れることで結合
  - 先頭の&はアドレス参照の&
  - 中間の&は
    - 1個目はビットAndの&
    - 2個目はアドレス参照の&
  - こうなるまで色々経緯があった
    - 最初は#define a (a_)式とか
      - operator()として認識される
    - $ を &apply にしようとか
      - そうすると数値とかが直接かけない

式

```
&print_& &apply& &take_& 10 &apply& map_(x --> 2*x) [1__]
```

- ラムダ式(のつもり)
  - -->演算子なんてない
  - operator--(後置デクリメント)とoperator>(Greater)の連結
    - C++はトークン分解するので
  - while(x --> 0){}とかたまにやります(xが0になるまで1ずつ減る)

- ちなみにこの括弧はmap_(変数)のoperator()(関数適用)

式

```
&print_& &apply& &take_& 10 &apply& map_(x --> 2*x) [1__]
```

- [1..]のつもり(似てる)(似てない)
- User-defined literals (udl) (C++11)
  - 数字や文字列の後に_付きのSuffixを置くと自分で処理を作れる
  - 42000_m == 42_km みたいなことができる
    - _mと_kmを定義
  - 返す型はなんでもいいのでユーザ定義で作ればいい
    - 今回はSeq<Int,Infinite>っていうのを返してます
- []は配列参照(operator[])です

式

```
&print_& &apply& &take_& 10 &apply& map_(x --> 2*x) [1__]
```

- とりあえずC++としてValidという話
- この時点で型が

式

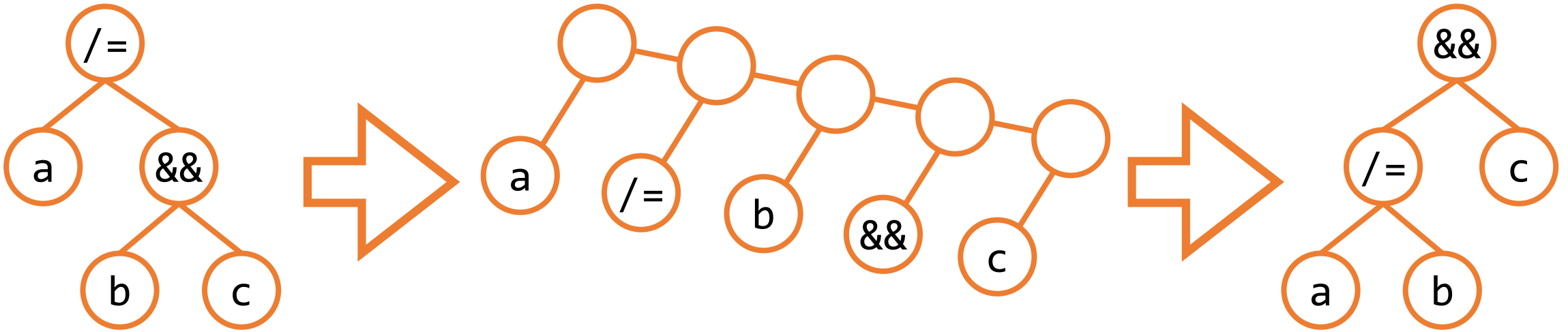
```
&print_& &apply& &take_& 10 &apply& map_(x --> 2*x) [1__]
```

- とりあえずC++としてValidという話
- この時点で型が

```
Seq<Seq<Seq<Seq<Seq<Seq<Seq<Value<Function<Print> >, Null>,
Seq<Apply, Null> >, Seq<Value<Function<Take> >, Null> >,
Value<Int> >, Seq<Apply, Null> >, Seq<Value<Function<Map> >,
Null> >, Seq<Seq<Seq<Seq<Seq<Variable<23>, Null>, Decrease>,
Greater>, Seq<Value<Int>, Seq<Multiply, Seq<Variable<23>, Null> >
> >, Seq<Seq<Bra, Seq<Int, Seq<Infinite, Null> > >, Ket> > >
```

式

- 正しい構文木をこの時点で生成するのは困難
  - a /= b && c == d は a /= (b && (c == d))になる
    - C++の/=は除算代入(Haskellでは否定条件演算子)
- 一度一列になおしてから構文解析する
  - 型レベルで

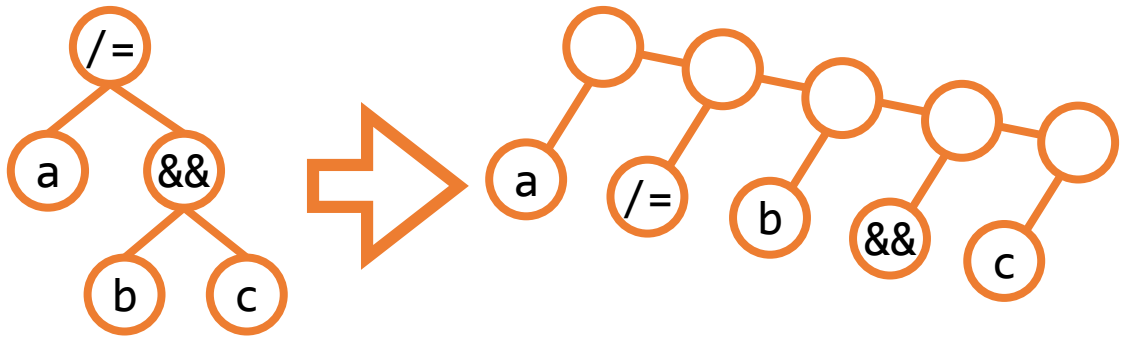


listify

```
template <略> auto listify(Seq<T,U> s, V v){
  listify(s.t,listify(s.u,v));
}
```

- Tの次にUが来る列をlistifyすると、
- Uをlistifyしたものの先頭にTをlistifyしたものがくる

- 実際こんなかんじ



tokenize

- ラムダ式用の `-->` や、冪乗の `**` などはそのままではだめ
- 一つのトークンとして再構成する必要がある

```
略 auto tokenize(Seq<Decrease,Seq<Greater,T>> s){
  return Seq<Lambda,略>(Lambda(),略);
}
```

- 走査して`--`と`>`の列を見つけたらLambdaだと思い込む
  - パターンマッチが簡単にかけるので処理も楽
- 略の中で後続のSeqに伝播している

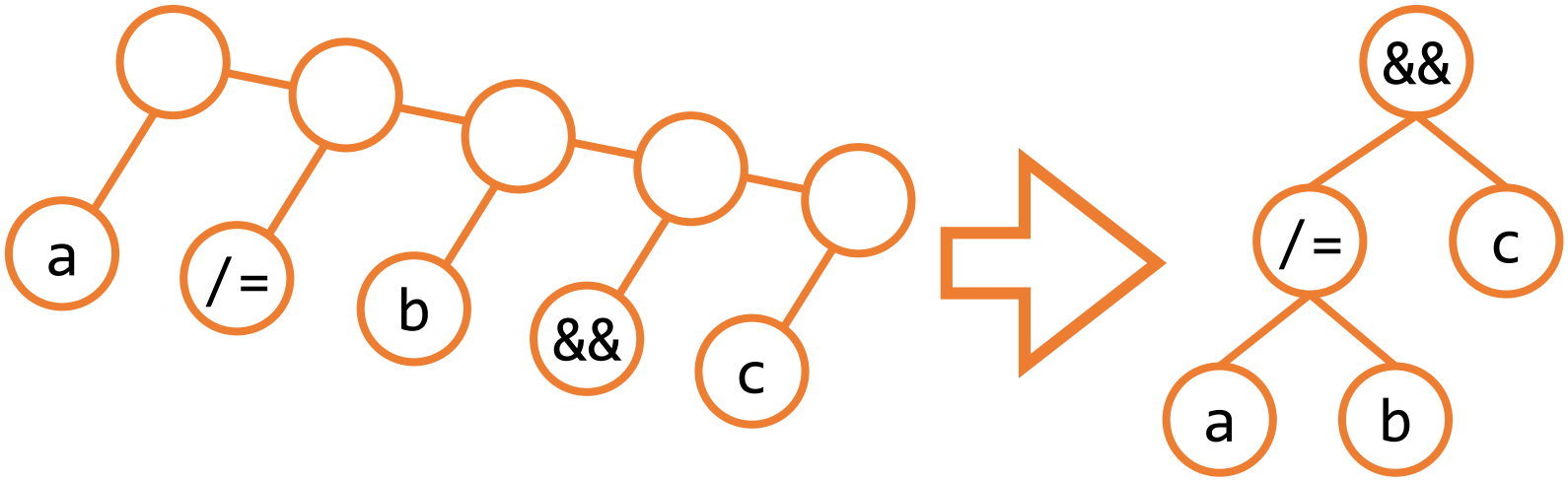
現在

```
Seq<Value<Function<Print> >, Seq<Apply,
Seq<Value<Function<Take> >, Seq<Value<Int>, Seq<Apply,
Seq<Value<Function<Map> >, Seq<Variable<23>, Seq<Lambda,
Seq<Value<Int>, Seq<Multiply, Seq<Variable<23>, Seq<Bra,
Seq<Int, Seq<Infinite, Seq<Ket, Null> > > > > > > > > > > > >
> > >
```

- 最後に > がいっぱい並んでいる
- SeqとSeqの間にそれぞれ1トークン
- Nullで終わる型レベルリスト

parse

- 正しい構文木を構築しなおす



parse

- BNF書いてー

```
Expr   ::= Expr1 | Expr $ Expr1
Expr1  ::= Expr2 | Expr1 >>= Expr2 | Expr1 >> Expr2
Expr2  ::= Expr3 | Expr2 || Expr3
Expr3  ::= Expr4 | Expr3 && Expr4
Expr4  ::= Expr5 | Expr4 < Expr5 | Expr4 > Expr5 | Expr4 <= Expr5 | Expr4 >= Expr5 | Expr4 == Expr5 | Expr4 /= Expr5
Expr5  ::= Expr6 | Expr5 ++ Expr6
Expr6  ::= Expr7 | Expr6 + Expr7 | Expr6 - Expr7
Expr7  ::= Expr8 | Expr7 * Expr8 | Expr7 / Expr8
Expr8  ::= Expr9 | Expr8 ^ Expr9 | Expr8 ** Expr9
Expr9  ::= let Decl in Expr | if Expr then Expr else Expr | Name --> Expr | Apply
Apply  ::= Factor | Apply Factor
Factor ::= Value | (Expr) | [] | [Int__] | [Expr(,Expr)*]
Decl   ::= (Name)+ = Expr
Value  ::= <primitive>
```

parse

- 再帰下降パーサ書いてー

```
略 auto expr(T t) {
  return expr_i(expr1(t));
}
略 auto expr_i(Pair<R,Seq<Apply,S>> t) {
  return Ret<Apply>()(t.t, expr(t.u.u));
}
略 auto expr_i(Pair<R,S> t) {
  return t;
}
//Expr ::= Expr1 | Expr $ Expr1
```

parse

- ぽん

```
Ope<Apply, Value<Function<Print> >, Ope<Apply, Ope<ApplyP,
Value<Function<Take> >, Value<Int> >, Ope<ApplyP,
Ope<ApplyP, <Value<Function<Map> >, LambdaA<23,
Ope<Multiply, Value<Int>, Variable<23> > > >, Inf> > >
```

parse

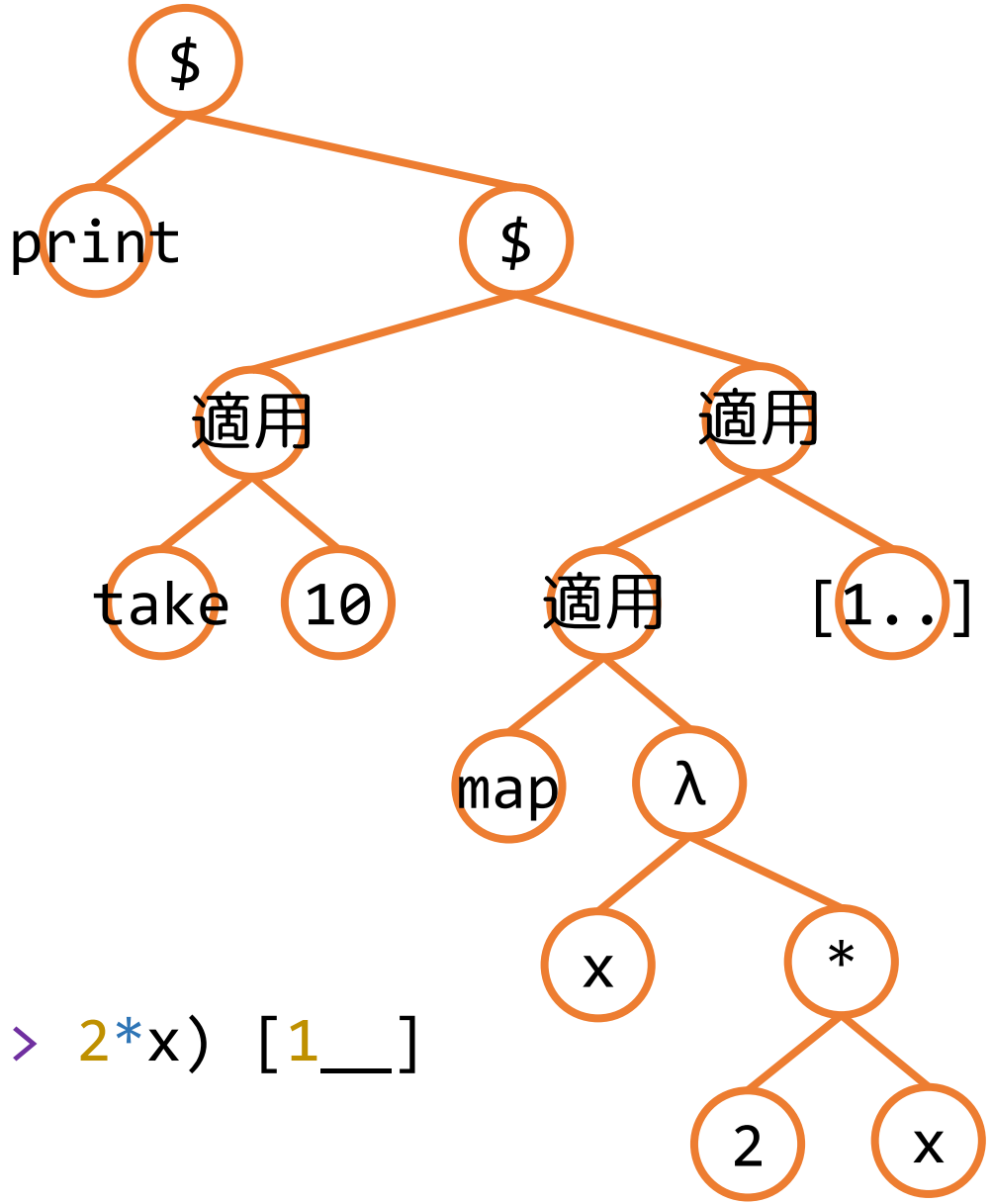
- ぽん

```
Ope<Apply, Value<Function<Print> >, Ope<Apply, Ope<ApplyP,
Value<Function<Take> >, Value<Int> >, Ope<ApplyP,
Ope<ApplyP, <Value<Function<Map> >, LambdaA<23,
Ope<Multiply, Value<Int>, Variable<23> > > >, Inf> > >
```

- Opeが演算子の部分(Applyは $、ApplyPは f xでの適用)

parse

- 見やすくするとこんなかんじ
- いわゆる抽象構文木
  - Abstract Syntax Tree



- まぁこう見れば何も怖くない

- 参考：元の形

```
print $ take 10 $ map_(x --> 2*x) [1__]
```

parse

- めんどくさかった
  - 状態を持てないのでPairを返して第二引数が残りのSeq
  - 一度右再帰で組んでから左再帰に直すみたいなことをした
  - $は右結合でf xの時は左結合とか
- そのかわりtemplateのパターンマッチがつよい
  - Seq<Variable<N>, S> とか
  - Pair<R, Seq<Infinite, Seq<Ket, S>>> とか
- 生C++よりtemplateのほうがいいのでは？？？

現状

- listify→tokenize→parse
- あとはEvaるだけ

現状

- listify→tokenize→parse
- あとはEvaるだけ

現状

- listify→tokenize→parse
- あとはEvaるだけ

- だけ？？？？？

Haskellとは

- 静的型付け
- 純粋関数型
- 遅延評価ベース
- グラフ簡約
- 再帰が当たり前

- な　言語

templateとは

- 静的型付けだがHaskellの型とは根本的に異なる
- 純粋関数型
- 遅延評価ベースなのは::valueなどを使った時だけ
- グラフ簡約などやるわけのない
- 再帰が当たり前なのは型が再帰してるときだけ

- という言語

かんたんにいうと

- 諦めました☆

かんたんにいうと

- 諦めました☆
  - 具体的には
    - 型付けと
    - グラフ簡約と
    - 遅延評価と
    - 再帰 の実装をやめました

かんたんにいうと

- 諦めました☆
  - 具体的には
    - 型付けと
    - グラフ簡約と
    - 遅延評価と
    - 再帰 の実装をやめました
  - 超ごめんなさい
  - わたしもめちゃくちゃつらい
    - こんなのHaskellじゃない
- 日程的にきつかったです
  - 作り始めたのは応募始まった4/20からですが忙しくて

いいわけ

- 型付けを行おうとするとHindley-Milner某アレをtemplate上で実装することになります
- つらいです

- C++ templateは一応型チェック自体はするので型がミスるとコンパイルエラーにはなります
  - 50行ぐらいエラー吐くんですけど
- 関数の型とかをちゃんと取得できないことが問題に・・・

いいわけ

- グラフ簡約は一度評価した値を次からも参照するみたいなやつです
- やるとしたら副作用が必須なので結構実装が重い

- だいたい型を持ってないのでその辺の処理ができない

いいわけ

- 遅延評価はグラフ簡約でできてます
  - できません

いいわけ

- template上で再帰しようとすると、止まりません
- ifの分岐どちらもinstance化するのでループ停止条件を作れません
- やるとしたら副作用付きで実世界で動かす必要があります

- 結果的に型付けをしなかったことで再帰ができなくなりました
  - つらい

というわけで

- 見た目重視、内部構造放置
- うごくものをつくることにする

- 今後の流れ
  - desugar
  - eval
  - 関数の実装
  - 結果

desugar

- 現在のletが扱いにくいのでいじる
  - `let f x y = z in e` は `let f = λx → λy → z in e` と等価
  - 適当にtemplateで処理
- ApplyPとApplyが分かれているので統一化など

- 本来は無限リスト([1..]など)の処理をする予定だった
  - `[x..]` ⇔ `let xs = x : map succ xs in xs`
  - 現在は100個生成するだけの超適当はりぼて実装

eval

- やるだけになる
  - 構文木を走査
  - Nodeにしたがって処理を分けてうごかす
- 多少の特殊処理が必要
  - let-in、ラムダ式
    - replace関数を作って変数名を全部式で変換
    - α変換してないので今でもその辺はつらい
      - そのうちde-Bruijn indexにでもします
- とりあえず簡単な式は動く

関数サポート

- tailとか、mapとか
  - 本来はnull/head/tailだけ作って再帰でいろいろ定義する予定だった
- それぞれ型は`Function<Tail>, Function<Map>`
  - そのまま持たせている
  - 融通が効く

- `Function`から`Map::operator()`とかを呼ぶ
  - メンバ関数なら型はtemplateで自由自在
  - 部分適用も別のファンクタを返すことで可能

例：tail (リストの先頭以外を返す)

```
struct Tail{
  略 List<T> operator ()(List<T> l) const {
    List<T> t = l;
    t.v.pop_back();
    return t;
  }
};
```

- 引数の型と戻り値の型を自分で指定しないと"何故か"動かない
  - あとconstが必要らしい(よくわかってない)
- とりあえずこれでapp(tail,list)みたいなのが動く
  - appが適用です

例：take (リスト先頭からn個を返す)

- 2引数必要なので、一度“適用済み関数”を返す
  - 部分適用？

```
struct Take{
  auto operator ()(Num i) const {
    return Function<TakeI>(TakeI(i));
  }
};
```

例：take (リスト先頭からn個を返す)

- 再度、適用先リストが与えられたら処理
  - 型が変わらないので楽

```
struct TakeI{
  Num i;
  略 List<T> operator ()(List<T> l) const {
    std::vector<T> t(l.v.begin(),l.v.begin() + i);
    return List<T>(t); //実際はendからとってたりする
  }
};
```

例：map (リスト全値にfを適用)

- かなり複雑(戻り値の型を決める機構が面倒)
- また値を返す代わりに別の関数を返す

```
struct Map{
  略 Function<MapI<F>> operator()(F f) const {
    return Function<MapI<F>>(MapI<F>(f));
  }
};
```

例：map (リスト全値にfを適用)

- 戻り値の型がわからないので別構造体に処理を投げる
  - 関数でもよかった

```
template <class F> struct MapI{
  F f;
  略 auto operator()(List<T> l) const -> 略 {
    return MapII<F,T,decltype(app(f,std::declval<T>()))>(f,l)();
  }
};
```

- declval(C++11)で値を拾って、適用結果の型を調べる
  - Nullの可能性があるので先頭拾ったりできない

例：map (リスト全値にfを適用)

- 渡された先では実際にリストを返す処理
  - 型がわかるのでリストを構築できる(List<X>)

```
template <class F, class T, class X> struct MapII{
  略;
  List<X> operator()() const {
    std::vector<X> v;
    for(auto e : t.v)v.push_back(app(f,e));
    return List<X>(v);
  }
};
```

そんなこんなで

```
main_= getLine >>= x -->
 _(putStrLn $ if_ x == “C++”
                then  “No more C++”
                else_ “Hello, “_+++x) >>
 _(print $ let f x = 2 * x in f 5) >>
 _(print $ take 10 $ map_(x --> 2*x) [1__]) >>
 _(print $ filter_(x --> x /= 0) [1_,0,8,0,0]) >>
  return_()
```

- 動きます。

実行部分

```
struct Hs{
  略 auto operator[](T t) -> 略 {
    return unwrap(eval(desugar(parse(tokenize(listify(t))))));
  }
}hs;
```

やったこと

- マクロ展開で出来るだけHaskellSyntaxをC++的にValidにする
- ExpressionTemplateでC++的構文木を生成
- 一度1列のトークン列に変換
- `-->` や `**` などのトークンをまとめる
- 自作HaskellSyntaxにもとづいて構文解析
- 多少の糖衣構文を潰す
- eval
  - 関数の処理を行う

ちなみに

- hs[<expr>]であることからわかるように、C++の値に変換できます。

```
std::cout << hs[let f x = 2 * x in f 5] << std::endl;
// 10

std::vector<int> v{1,2,4,7};
v = hs[map_(x --> x^2) $ v];
// v = {1,4,16,49}
```

余談(1)

余談(1)

```
21 clang          0x083f5126 clang::CodeGen::CodeGenModule::EmitGlobalFunctionDefinition(clang::GlobalDecl) + 454
22 clang          0x083f54c0 clang::CodeGen::CodeGenModule::EmitGlobalDefinition(clang::GlobalDecl) + 320
23 clang          0x083f5ce3 clang::CodeGen::CodeGenModule::EmitGlobal(clang::GlobalDecl) + 1347
24 clang          0x083f672c
25 clang          0x083d4f44
26 clang          0x083d14dd
27 clang          0x0825307e clang::ParseAST(clang::Sema&, bool, bool) + 350
28 clang          0x0825fa18 clang::ASTFrontendAction::ExecuteAction() + 120
29 clang          0x083d0ce1 clang::CodeGenAction::ExecuteAction() + 33
30 clang          0x0825ff48 clang::FrontendAction::Execute() + 168
31 clang          0x08241255 clang::CompilerInstance::ExecuteAction(clang::FrontendAction&) + 261
32 clang          0x08229ed7 clang::ExecuteCompilerInvocation(clang::CompilerInstance*) + 1463
33 clang          0x082247c8 cc1_main(char const**, char const**, char const*, void*) + 856
34 clang          0x082235d9 main + 7305
35 libc.so.6      0xb64943c6 __libc_start_main + 230
36 clang          0x0822425d
Stack dump:
0.      Program arguments: /usr/bin/clang -cc1 -triple i386-pc-linux-gnu -emit-obj -mrelax-all -disable-free -disable-llvm-verifier -main-file-name mai
n.cpp -mrelocation-model static -mdisable-fp-elim -fmath-errno -masm-verbose -mconstructor-aliases -target-cpu pentium4 -target-linker-version 2.21.1 -
resource-dir /usr/bin/../lib/clang/3.3 -cxx-isystem /usr/local/include -cxx-isystem /usr/local/include/c++/4.8.0 -cxx-isystem /usr/local/include/c++/4.
8.0/i686-pc-linux-gnu -cxx-isystem /usr/local/include/c++/4.8.0/backward -internal-isystem /usr/lib/gcc/i686-pc-linux-gnu/4.5.3/include/g++-v4 -interna
l-isystem /usr/lib/gcc/i686-pc-linux-gnu/4.5.3/include/g++-v4/i686-pc-linux-gnu -internal-isystem /usr/lib/gcc/i686-pc-linux-gnu/4.5.3/include/g++-v4/b
ackward -internal-isystem /usr/local/include -internal-isystem /usr/bin/../lib/clang/3.3/include -internal-externc-isystem /include -internal-externc-i
system /usr/include --std=c++1y -fdeprecated-macro -fdebug-compilation-dir /home/phi16/Source/C++/Haskell -ferror-limit 19 -fmessage-length 0 -mstackre
align -fobjc-runtime=gcc -fobjc-default-synthesize-properties -fcxx-exceptions -fexceptions -fdiagnostics-show-option -backend-option -vectorize-loops
-o /tmp/main-IszSk8.o -x c++ main.cpp
1.      <eof> parser at end of file
2.      main.cpp:14:5: LLVM IR generation of declaration 'main'
3.      main.cpp:14:5: Generating code for declaration 'main'
clang: error: unable to execute command: Segmentation fault
clang: error: clang frontend command failed due to signal (use -v to see invocation)
clang version 3.3 (tags/RELEASE_33/final)
Target: i386-pc-linux-gnu
Thread model: posix
clang: note: diagnostic msg: PLEASE submit a bug report to http://llvm.org/bugs/ and include the crash backtrace, preprocessed source, and associated r
un script.
clang: note: diagnostic msg:
********************

PLEASE ATTACH THE FOLLOWING FILES TO THE BUG REPORT:
Preprocessed source(s) and associated run script(s) are located at:
clang: note: diagnostic msg: /tmp/main-42mlEC.cpp
```

余談(1)

```
21 clang          0x083f5126 clang::CodeGen::CodeGenModule::EmitGlobalFunctionDefinition(clang::GlobalDecl) + 454
22 clang          0x083f54c0 clang::CodeGen::CodeGenModule::EmitGlobalDefinition(clang::GlobalDecl) + 320
23 clang          0x083f5ce3 clang::CodeGen::CodeGenModule::EmitGlobal(clang::GlobalDecl) + 1347
24 clang          0x083f672c
25 clang          0x083d4f44
26 clang          0x083d14dd
27 clang          0x0[illegible]7e clang::ParseAST(clang::Sema&, bool, bool) + 350
28 clang          0x0825fa18 clang::ASTFrontendAction::ExecuteAction() + 120
29 clang          0x083d0ce1 clang::CodeGenAction::ExecuteAction() + 33
30 clang          0x0825ff48 clang::FrontendAction::Execute() + 168
31 clang          0x08241255 clang::CompilerInstance::ExecuteAction(clang::FrontendAction&) + 261
32 clang          0x08229ed7 clang::ExecuteCompilerInvocation(clang::CompilerInstance*) + 1463
33 clang          0x082247c8 cc1_main(char const**, char const**, char const*, void*) + 856
34 clang          0x082235d9 main + 7305
35 libc.so.6      0xb64943c6 __libc_start_main + 230
36 clang          0x0822425d
Stack dump:
0.      Program arguments: /usr/bin/clang -cc1 -triple i386-pc-linux-gnu [illegible] -disable-free -disable-llvm-verifier -main-file-name main.cpp -mrelocation-model static -mdisable-[illegible] -fmath-errno -masm-verbose -mconstructor-aliases -target-cpu pentium4 -target-linker-version 2.21.1 -resource-dir /usr/bin/../lib/clang/3.3 -cxx-isystem /usr/local/include -cxx-isystem /usr/local/include/c++/4.8.0 -cxx-isystem /usr/local/include/c++/4.8.0/i686-pc-linux-gnu -cxx-isystem /usr/local/include/c++/4.8.0/backward -internal-isystem /usr/lib/gcc/i686-pc-linux-gnu/4.5.3/include/g++-v4 -internal-isystem /usr/lib/gcc/i686-pc-linux-gnu/4.5.3/include/g++-v4/i686-pc-linux-gnu -internal-isystem /usr/lib/gcc/i686-pc-linux-gnu/4.5.3/include/g++-v4/backward -internal-isystem /usr/local/include -internal-isystem /usr/bin/../lib/clang/3.3/include -internal-externc-isystem /include -internal-externc-isystem /usr/include -std=c++11 -[illegible] -fdeprecated-macro -fdebug-compilation-dir /home/phi16/Source/C++/Haskell -ferror-limit 19 -fmessage-length 0 -mstackrealign -fobjc-runtime=gcc -fobjc-default-synthesize-properties -fcxx-exceptions -fexceptions -fdiagnostics-show-option -backend-option -vectorize-loops -o /tmp/main-IszSk0.o -x c++ main.cpp
1.      <eof> parser at end of file
2.      main.cpp:14:5: LLVM IR generation of declaration 'main'
3.      main.cpp:14:5: Generating code for declaration 'main'
clang: error: unable to execute command: Segmentation fault
clang: error: clang frontend command failed due to signal (use -v to see invocation)
clang version 3.3 (tags/RELEASE_33/final)
Target: i386-pc-linux-gnu
Thread model: posix
clang: note: diagnostic msg: PLEASE submit a bug report to http://llvm.org/bugs/ and include the crash backtrace, preprocessed source, and associated run script.
clang: note: diagnostic msg:
********************

PLEASE ATTACH THE FOLLOWING FILES TO THE BUG REPORT:
Preprocessed source(s) and associated run script(s) are located at:
clang: note: diagnostic msg: /tmp/main-42mlEC.cpp
```

> Clang3.3が <

> Internal Compiler Error <

余談(1)

> Clang3.3が <
>Internal Compiler Error<

- でも色々いじってたら治りました
- auto周りが不安定みたい
  - 今回voidを渡してしまっていたのでそれのせいかと

余談(1)

- でも何か根本的にやばいのではないかとおもってHEADをビルド
- 通らない

- `undefined reference to `std::string::pop_back()`

- ！？！？！？
- よくわからないので放置しています

- なんであれClang3.3で動いたのでいいです
  - gccは知らない

余談(2)

Numのoperator <<を組んでいたら何故か動かない

- `std::cout << hs[5_] << std::endl`

```
std::ostream& operator <<(std::ostream& os, Num x){
  os << x.x;
  return os;
}
```

- `std::cout << hs[5_].x << std::endl`は動く
  - ！？！？
- バグ挙動っぽい・・・？

clangのバグな感じありますね 1:03

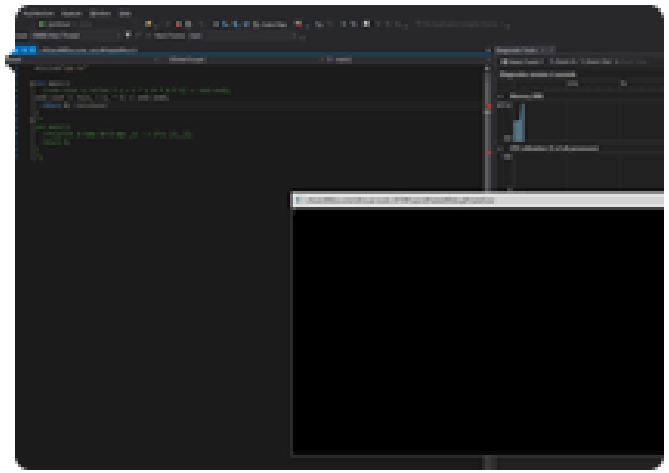
詰んだな 1:04

clangのバグな感じありますね

1:03

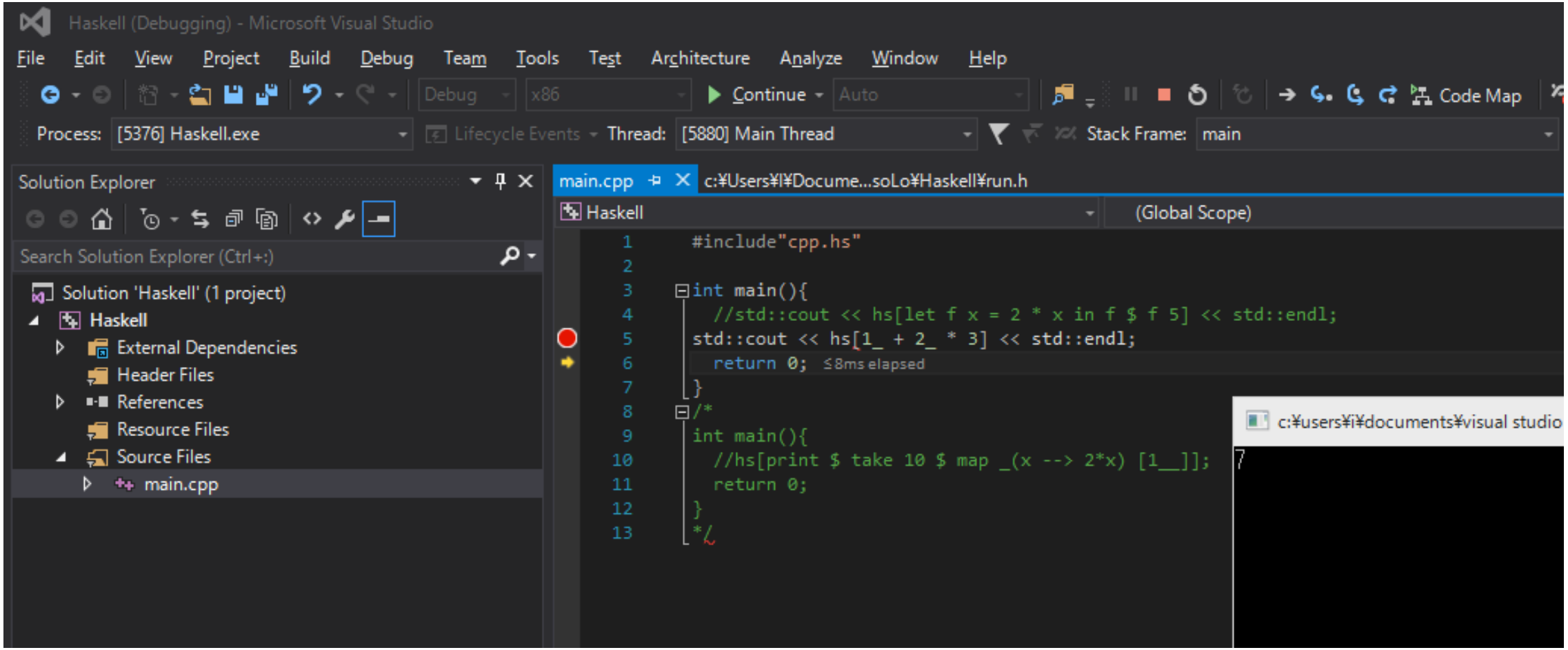
詰んだな

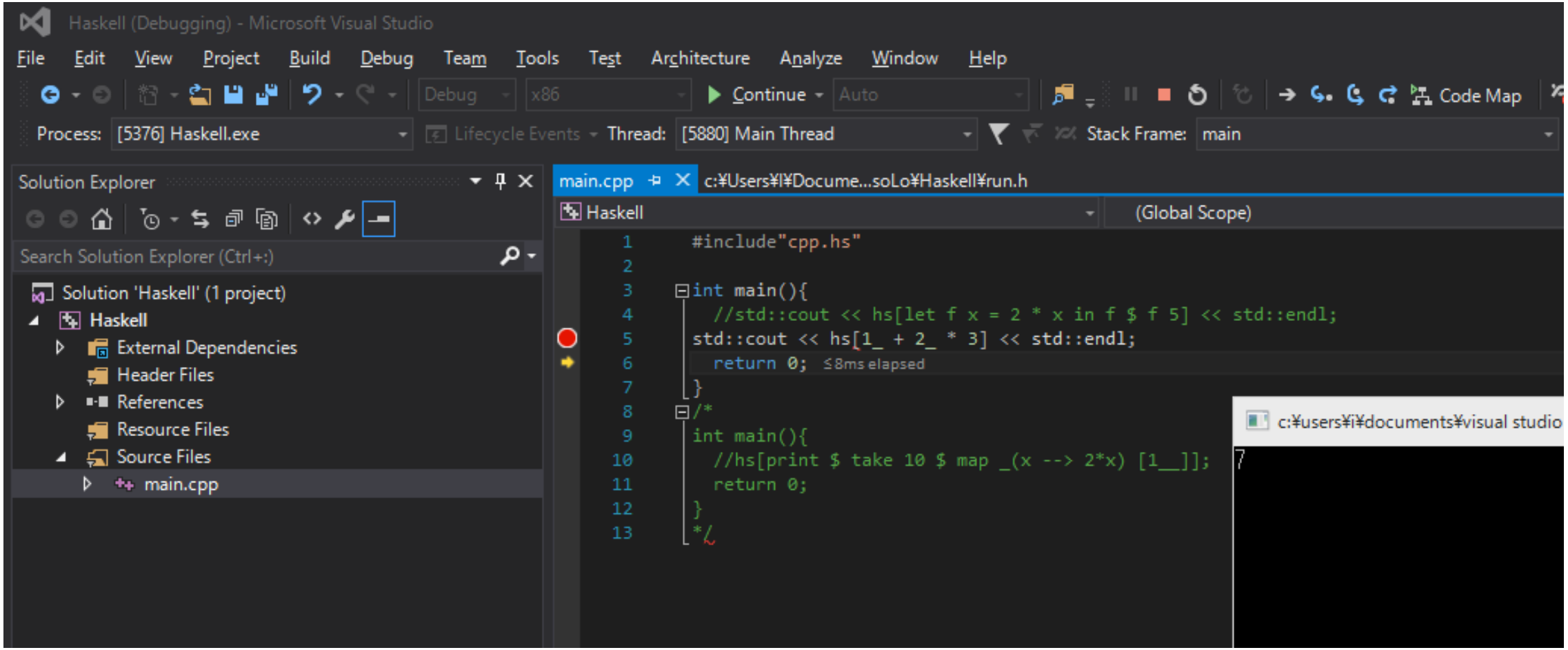
1:04



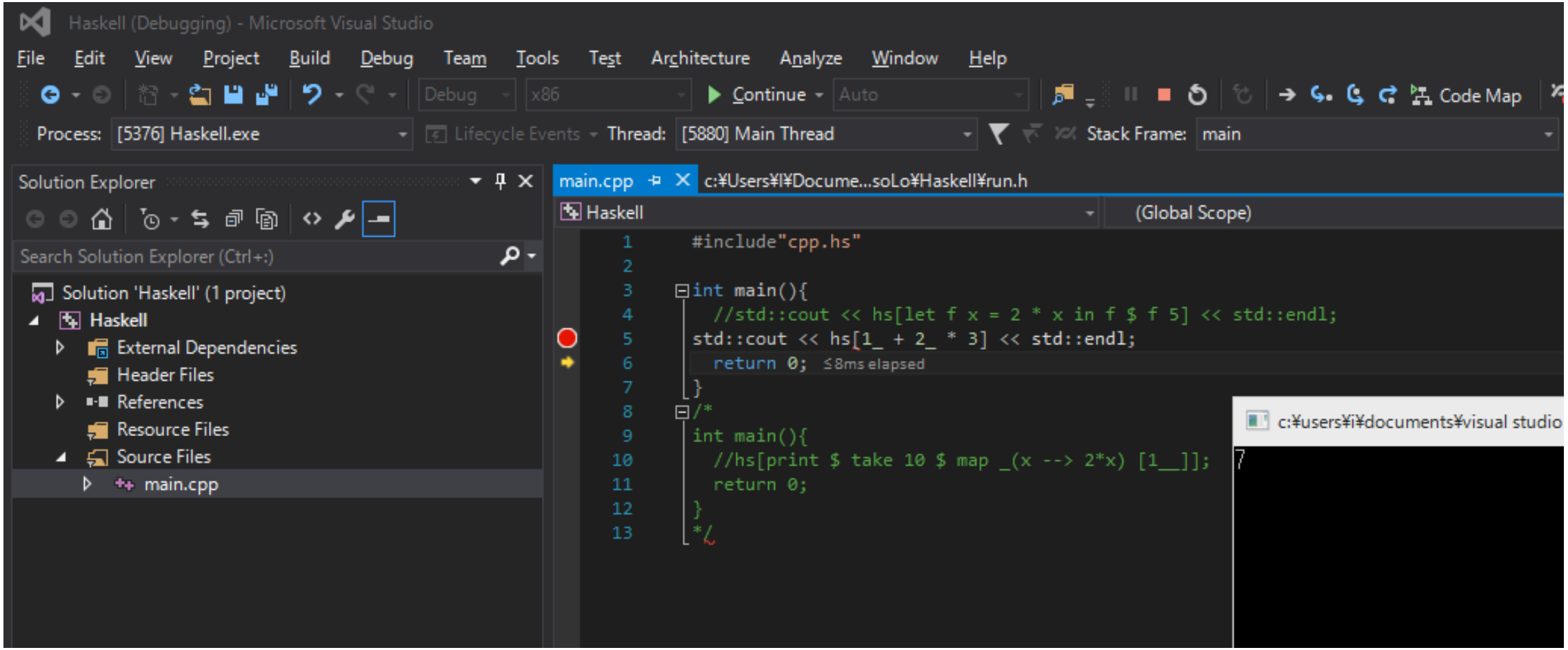
1:04

ファイルを受信しました。


Haskell (Debugging) - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Architecture Analyze Window Help
Debug
x86
Continue
Auto
Code Map
Process: [5376] Haskell.exe
Lifecycle Events
Thread: [5880] Main Thread
Stack Frame: main
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Haskell' (1 project)
Haskell
External Dependencies
Header Files
References
Resource Files
Source Files
main.cpp
main.cpp
c:¥Users¥I¥Docume...soLo¥Haskell¥run.h
Haskell
(Global Scope)
1 #include"cpp.hs"
2
3 int main(){
4 //std::cout << hs[let f x = 2 * x in f $ f 5] << std::endl;
5 std::cout << hs[1_ + 2_ * 3] << std::endl;
6 return 0; ≤8ms elapsed
7 }
8 /*
9 int main(){
10 //hs[print $ take 10 $ map _(x --> 2*x) [1__]];
11 return 0;
12 }
13 */
c:¥users¥i¥documents¥visual studio
7



- ＞VisualStudioで動いた＜
  - VS2015 CTP 6



- ＞VisualStudioで動いた＜
  - VS2015 CTP 6
- これからはVSの時代

operator<<の方でstd::flushすると動く

1:10

余談(2)

```
std::ostream& operator <<(std::ostream& os, Num x){
  os << x.x << std::flush;
  return os;
}
```

余談(2)

```
std::ostream& operator <<(std::ostream& os, Num x){
  os << x.x << std::flush;
  return os;
}
```

- Clangはストリーム周りのバグがたまにあるらしい

余談(3)

- C++からHaskellが使えるようになった

余談(3)

- C++からHaskellが使えるようになった
- HaskellからC++を使おう！

余談(3)

- C++からHaskellが使えるようになった
- HaskellからC++を使おう！
  - ※書きたいわけではない

余談(3)

- C++からHaskellが使えるようになった
- HaskellからC++を使おう！
  - ※書きたいわけではない
- TemplateHaskell
- QuasiQuotes
- あとは適当にClangを呼ぶ
  - ずるいとかいわない

余談(3)

```haskell
import Cpp

main :: IO ()
main = do
  print $ ([cpp| std::vector<int>({1,2,3}) |] :: [Int])
  // [1,2,3]
```

- 30分でできた
  - こっちのほうが本当の ＋コンパイル時計算＋ してるので
  - コンパイル時 コンパイル&実行

  - 反クォートは使ってないです

まとめ

- C++でHaskellっぽく書ける！
  - HaskellでもC++っぽく書ける！

- C++ templateは強い
  - TemplateHaskellはもっと強い

- Haskellを書けた人生だった
    - (Haskellっぽいだけ)
  - C++14を書ける人生だった
    - decltype省略しあわせ
    - (C++が書けるとは言ってない)

ありがとうございました

- 提供
- コンパイラ：Clang 3.3
- バグ修正等：@wx257osn2
  - ありがとうございました

ありがとうございました

質問あればどぞ

- 提供
- コンパイラ：Clang 3.3
- バグ修正等：@wx257osn2
  - ありがとうございました