

自己紹介

- h_sakurai
- 仙台出身
- Scalaでコンパイラ作ろうとしてます
- CyberXでソーシャルアプリ作ってます

MAKING OF LOW LEVEL NEW NATIVE COMPILER

H I R O S H I S A K U R A I

アジェンダ

＊ マーケティング

＊ インタプリタ

＊ どう新しくする？

＊ マクロ

＊ コンパイルインフラス
トラクチャ

＊ コンパイラ

＊ パーサ(構文解析器)

＊ 今後

マーケティング

市場を調査して必要な物を把握する

今ある言語

- ◆ Perl,Ruby,PHP,Python 等LLは供給過多
- ◆ Java,C#,ActionScriptは十分新しい
- ◆ C++,C,Objective C が古くさい
- ◆ VisualBasic6.0 が古くさい
- ◆ C言語の置き換えが必要そう

なぜCは変わらなかった？

- ◆ 最初の10年は構造化、次はオブジェクト指向、ここ10年はVM,GC,LL志向
- ◆ 言語はどんどん便利に高級になった
- ◆ CPUはどんどん速くなってきた
- ◆ C言語は偉大すぎた

現在

- ◆ Windowsは.net frameworkのみで構築されることはなかった。VM、GC重い
- ◆ CPUのクロックアップは鈍り、マルチコアの時代になりました
- ◆ LLは高速化の時代。Cでライブラリを書く人が少ない、書きたくない

C言語,C++,ObjectiveC

- ◆ C言語はUnix
- ◆ C++はWindows
- ◆ ObjectiveCはMac OS
- ◆ CはOSを支える

D,golang

- ◆ GC付きネイティブ言語
- ◆ GCがあると、OSを作れない
- ◆ PHP,Ruby,Pythonのライブラリを作れ
ない

必要な言語

- ◆ OSが作れる
- ◆ LLのライブラリを奇麗に作れる
- ◆ 言語が奇麗に作れる
- ◆ GCが不要、メモリ管理を奇麗にできる
- ◆ 新しくて使いやすいC言語の代わり

アジェンダ

＊ マーケティング

＊ インタプリタ

＊ どう新しくする？

＊ マクロ

＊ コンパイルインフラス
トラクチャ

＊ コンパイラ

＊ パーサ(構文解析器)

＊ 今後

どう新しくする?
c 言語を超えるために

新しくする箇所

- CのマクロをLisp級マクロにする
- マッチング構文を入れる
- 多値(tuple 無名構造体),List,クロージャ
- 関数型言語で美しく簡単な実装を作る
- コンパイルインフラストラクチャを作る

Lisp級マクロ

- C言語のマクロは置換が主だった
- Lispのマクロは置換プログラムを作って使える
- プログラムがデータだから可能
- S式(Listを表す式)の代わりの式(C式)を導入する
- 好きに構文追加できるようになる

C式 S式の代替

- LispのS式の代わりのC言語風式言語で私が考えた
- C like expression, Compact expressionの略
- `f (a(1,2,3)==5) b+c; else {e=c+d[2][6];}`
のような記述が可能
- 内部データは多値(tuple)で表現
- C式`if(a) b else c`は 多値 `("if", "(", ", "a", ",")", ("b", "else", "c"))`

マクロの功罪

- マクロは混乱をもたらす
- 高レベルな言語なら動的な仕組みがあるので不要？
- コンパイル時に静的に解決するので高速化に使える
- 新しい構文を追加することで、記述を奇麗に出来る
- より低レベルな言語にはあると便利な機能

コンパイラを美しく書こう

- コンパイラを奇麗で短く美しく書こう
- 奇麗で短ければ簡単理解出来る
- 簡単なので、各プラットフォームに展開しやすい
- 発展すればみんな使うだろう

どう簡単にする？

- 関数型言語で書けば奇麗に書ける。
- でもいつも使ってる言語と全然違と忘れる。
- C言語風言語で関数型言語である言語を使おう。
- 今のところScalaが最適だからScalaを使おう。
- いずれ、開発した言語に置き換えよう。

関数型言語の使える機能

- マッチング構文(switchの凄い奴)
正規表現,多値やクラス,リストでもswitchできる
値のバインディング機能でスッキリ書ける
- 多値(tuple)、関数内関数、再帰関数、クロージャ
- map,foldLeft等の高階関数も短く書けるポイント

SOURCE CODE

Scalaの例

```
// 多値とパターンマッチ, バインディングによる計算機

object calc {
  def exec(a:Any):Int = a match {
    case (a,"+",b) => exec(a)+exec(b)
    case (a,"*",b) => exec(a)*exec(b)
    case a:Int => a
  }
  def main(argv:Array[String]) {
    println(exec(1,"+",(2,"*",3)))
  }
}

// php
function e($a) {
  switch (true) {
  case is_array($a) && count($a)==3 && $a[1]=="+": return e($a[0])+e($a[2]);
  case is_array($a) && count($a)==3 && $a[1]=="-": return e($a[0])*e($a[2]);
  default: return a;
  }
}
echo e(array(1,"+",array(2,"*",3)))."\n";
```

```
// 高階関数map

object maptest {
  def main(argv:Array[String]) {
    println(List(1,2,3).map(_+1))
  }
}

// List(2,3,4)
```

```
// Listによるマッチングと関数内関数

object maptest {
  def main(argv:Array[String]) {
    def addOne(a>List[Int]):List[Int] = a match {
      case List() => List()
      case x::xs => (x+1)::addOne(xs)
    }
    println(addOne(List(1,2,3)))
  }
}

// List(2,3,4)
```

```
// case classによるマッチング

object calc2 {
  def main(args:Array[String]) {
    println(exec(Add(Val(1),Mul(Val(2),Val(3)))))
  }
  sealed abstract class Atom
  case class Add(l:Atom,r:Atom) extends Atom
  case class Mul(l:Atom,r:Atom) extends Atom
  case class Val(a:Int) extends Atom
  def exec(a:Atom):Int = a match {
    case Add(a,b) => exec(a)+exec(b)
    case Mul(a,b) => exec(a)*exec(b)
    case Val(a) => a
  }
}
```

```
// 正規表現によるマッチング
// 型推論と書き換え不能な変数val

object regmatch {
  def main(argv:Array[String]) {
    val reg = """(a+)(b+)(a+)""".r
    val src = "aaaabbbaaaa"
    src match {
      case reg(a,b,c) => println("ok")
      case _ => println("ng")
    }
  }
}
```

// 多値とパターンマッチ, バインディングによる計算機

```
object calc {  
    def exec(a:Any):Int = a match {  
        case (a,"+",b) => exec(a)+exec(b)  
        case (a,"*",b) => exec(a)*exec(b)  
        case a:Int => a  
    }  
    def main(argv:Array[String]) {  
        println(exec(1,"+",(2,"*",3)))  
    }  
}
```

// php

```
function e($a) {  
    switch (true) {  
        case is_array($a) && count($a)==3 && $a[1]=="+": return e($a[0])+e($a[2]);  
        // E  
        case is_array($a) && count($a)==3 && $a[1]=="-": return e($a[0])*e($a[2]);  
        default: return $a;  
    }  
}  
// L  
echo e(array(1,"+",array(2,"*",3)))."\n";
```

.. --,,--,,

Scalaの例

```
// 高階関数map
object maptest {
  def main(argv:Array[String]) {
    println(List(1,2,3).map(_+1))
  }
}
// List(2,3,4)
```

```
// Listによるマッチングと関数内関数
object maptest {
  def main(argv:Array[String]) {
    def addOne(a>List[Int]):List[Int] = a match {
      case List() => List()
      case x::xs => (x+1)::addOne(xs)
    }
    println(addOne(List(1,2,3)))
  }
}
// List(2,3,4)
```

```
// case classによるマッチング
object calc2 {
  def main(args:Array[String]) {
    println(exec(Add(Val(1),Mul(Val(2),Val(3)))))
  }
  sealed abstract class Atom
  case class Add(l:Atom,r:Atom) extends Atom
  case class Mul(l:Atom,r:Atom) extends Atom
  case class Val(a:Int) extends Atom
  def exec(a:Atom):Int = a match {
    case Add(a,b) => exec(a)+exec(b)
    case Mul(a,b) => exec(a)*exec(b)
    case Val(a) => a
  }
}
```

```
// 正規表現によるマッチング
// 型推論と書き換え不能な変数val
object regmatch {
  def main(args:Array[String]) {
    val reg = """(a+)(b+)(a+)""".r
    val src = "aaaabbbaaaa"
    src match {
      case reg(a,b,c) => println("ok")
      case _ => println("ng")
    }
  }
}
```

Scalaの例

```
// 高階関数map

object maptest {
  def main(argv:Array[String]) {
    println(List(1,2,3).map(_+1))
  }
}

// List
object addOne {
  def addOne(xs: List[Int]): List[Int] = xs match {
    case List() => List()
    case x::xs => (x+1)::addOne(xs)
  }
  println(addOne(List(1,2,3)))
}

// List(2,3,4)
```

```
// case classによるマッチング

object calc2 {
  def main(args:Array[String]) {
    println(exec(Add(Val(1),Mul(Val(2),Val(3)))))
  }
  sealed abstract class Atom
  case class Add(l:Atom,r:Atom) extends Atom
  case class Mul(l:Atom,r:Atom) extends Atom
  case class Val(a:Int) extends Atom
  val(a:Atom):Int = a match {
    case Add(l,r) => exec(l)+exec(r)
    case Mul(l,r) => exec(l)*exec(r)
    case Val(a) => a
  }
}
```

!表現によるマッチング
論と書き換え不能な変数val

```
regmatch {
  main(args:Array[String]) {
    reg = """(a+)(b+)(a+)""".r
    src = "aaaabbbaaaa"
    match {
      case reg(a,b,c) => println("ok")
      case _ => println("ng")
    }
  }
}
```

Scalaの例

```
// Listによるマッチングと関数内関数
object maptest {
  def main(argv:Array[String]) {
    def addOne(a>List[Int]):List[Int] = a match {
      case List() => List()
      case x::xs => (x+1)::addOne(xs)
    }
    println(addOne(List(1,2,3)))
  }
}
// List(2,3,4)
```

```
// case classによるマッチング
object calc2 {
  def main(args:Array[String]) {
    println(exec(Add(Val(1),Mul(Val(2),Val(3)))))
  }
  sealed abstract class Atom
  case class Add(l:Atom,r:Atom) extends Atom
  case class Mul(l:Atom,r:Atom) extends Atom
  case class Val(a:Int) extends Atom
  def exec(a:Atom):Int = a match {
    case Add(a,b) => exec(a)+exec(b)
    case Mul(a,b) => exec(a)*exec(b)
    case Val(a) => a
  }
}
```

```
// 正規表現によるマッチング
// 型推論と書き換え不能な変数val
object regmatch {
  def main(args:Array[String]) {
    val reg = """(a+)(b+)(a+)""".r
    val src = "aaaabbbaaaa"
    src match {
      case reg(a,b,c) => println("ok")
      case _ => println("ng")
    }
  }
}
```

Scalaの例

// Listによるマッチングと関数内関数

```
object maptest {
  def main(argv:Array[String]) {
    def addOne(a>List[Int]):List[Int] = a match {
      case List() => List()
      case x::xs => (x+1)::addOne(xs)
    }
    println(addOne(List(1,2,3)))
  }
}
// List(2,3,4)
```

```
// case classによるマッチング
object calc2 {
  def main(args:Array[String]) {
    println(exec(Add(Val(1),Mul(Val(2),Val(3)))))
  }
  sealed abstract class Atom
  object Add extends Atom
  object Mul extends Atom
  object Val extends Atom
  trait Executable {
    def exec(a:Atom):String
  }
  object Executable {
    def apply(a:Atom):Executable = a match {
      case Add(a,b) => ()+exec(a)+exec(b)
      case Mul(a,b) => ()*exec(a)+exec(b)
    }
  }
}
```

チング
能な変数val

ring]) {
'a+')""".r
a"

rintln("ok")
ig")

Scalaの例

```
// case classによるマッチング
object calc2 {
  def main(args:Array[String]) {
    println(exec(Add(Val(1),Mul(Val(2),Val(3)))))
  }
  sealed abstract class Atom
  case class Add(l:Atom,r:Atom) extends Atom
  case class Mul(l:Atom,r:Atom) extends Atom
  case class Val(a:Int) extends Atom
  def exec(a:Atom):Int = a match {
    case Add(a,b) => exec(a)+exec(b)
    case Mul(a,b) => exec(a)*exec(b)
    case Val(a) => a
  }
}
```

```
// 正規表現によるマッチング
// 型推論と書き換え不能な変数val
object regmatch {
  def main(args:Array[String]) {
    val reg = """(a+)(b+)(a+)""".r
    val src = "aaaabbbaaaa"
    src match {
      case reg(a,b,c) => println("ok")
      case _ => println("ng")
    }
  }
}
```

Scalaの例

```
// case classによるマッチング

object calc2 {
  def main(args:Array[String]) {
    println(exec(Add(Val(1),Mul(Val(2),Val(3)))))
  }
  sealed abstract class Atom
  case class Add(l:Atom,r:Atom) extends Atom
  case class Mul(l:Atom,r:Atom) extends Atom
  case class Val(a:Int) extends Atom
  def exec(a:Atom):Int = a match {
    case Add(a,b) => exec(a)+exec(b)
    case Mul(a,b) => exec(a)*exec(b)
    case Val(a) => a
  }
}
```

マッチング
不能な変数val
String] {
)(a+)""".r
aa"
println("ok")
'ng')

Scalaの例

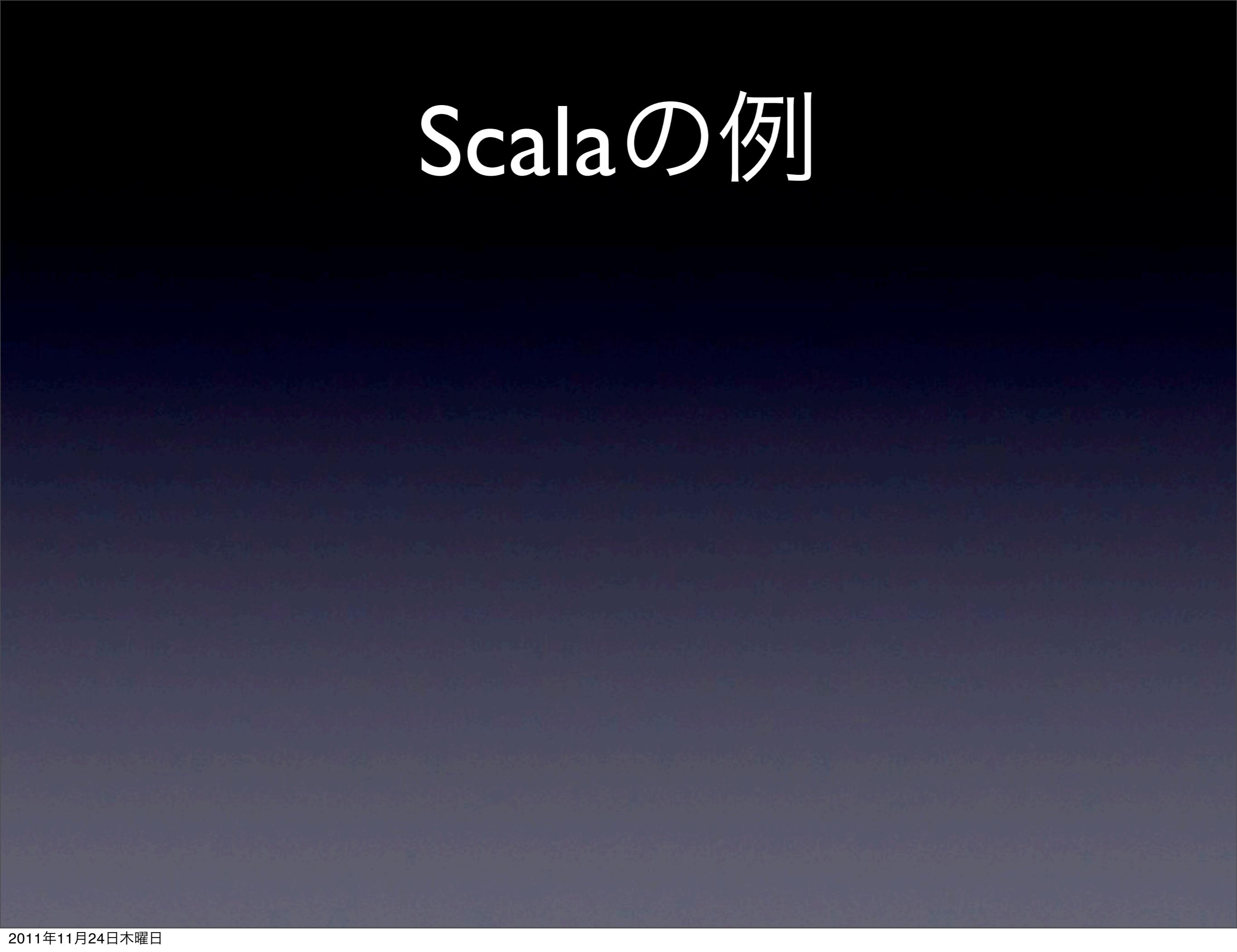
```
// 正規表現によるマッチング
// 型推論と書き換え不能な変数val
object regmatch {
  def main(args:Array[String]) {
    val reg = """(a+)(b+)(a+)""".r
    val src = "aaaabbbaaaa"
    src match {
      case reg(a,b,c) => println("ok")
      case _ => println("ng")
    }
  }
}
```

Scalaの例

```
// 正規表現によるマッチング
// 型推論と書き換え不能な変数val

object regmatch {
    def main(args:Array[String]) {
        val reg = """(a+)(b+)(a+)""".r
        val src = "aaaabbbaaaa"
        src match {
            case reg(a,b,c) => println("ok")
            case _ => println("ng")
        }
    }
}
```

Scalaの例



作り方の文書を作る

- たとえ簡単に作れても、自分だけ簡単ではどうしようもない。
- 簡単に書く方法をわかりやすく書いて広めよう
- 簡単に楽しくコンパイラが作れたらいいよね
- 開発者を育てて、展開してもらおう

これで言語は発展し続ける

- Lispは古い言語だがいまも発展し続けている
- S式の採用がLispが発展し続けている肝だ
- 根本的な改革が、C式により実現される
- この改革が成功すれば次々と新しい技術を生み出して行けるはずだ
- だから、これだけの機能があれば十分なのだ

さらに発展させたい

- C言語はC言語ファミリと呼べる派生言語が生まれた
- C++, Objective C, Perl, PHP, Java, C#, D
JavaScript, ActionScript, Haxe, Scala等多岐にわたる
- GCCはC言語だけではなく、多様な言語をコンパイル出来て、多様なCPU、OS用にコンパイル出来る
- コンパイラを作るプラットフォームを作りたい

アジェンダ

＊ マーケティング

＊ インタプリタ

＊ どう新しくする？

＊ マクロ

＊ コンパイルインフラス
トラクチャ

＊ コンパイラ

＊ パーサ(構文解析器)

＊ 今後

コンパイラ
インフラストラクチャ
コンパイラを作る基盤技術

コンパイルインフラストラクチャ

- * コンパイルインフラストラクチャとはコンパイラを作る基盤技術のこと
- * GCC GNUのCコンパイラからの派生物
- * LLVM flashに出力もできる新しい物
- * .net framework マイクロソフトのVM上で動作
- * COINS 国産Java製のオブジェクト指向等がある

関数型言語による インフラストラクチャを作ろう

- ※ 関数型言語によるインフラストラクチャは少ない
- ※ 関数型言語を使った方が奇麗に書ける
- ※ 関数型言語は研究目的だった
- ※ 実用的なコンパイラもあり、実用段階まで来てる
- ※ Scalaでコンパイルインフラストラクチャを作ろう！！そして、オリジナル言語に置き換えよう

アジェンダ

＊ マーケティング

＊ インタプリタ

＊ どう新しくする？

＊ マクロ

＊ コンパイルインフラス
トラクチャ

＊ コンパイラ

＊ パーサ(構文解析器)

＊ 今後

パーサ

ソースコードから構造を解析する
もの

Parser 構文解析器

- パーサは文字列から構造を作り出す物
- 構文解析には上昇型と下降型がある
- 上昇型は手書きで書くのは難しい
- 下降型は手書きできるので簡単

下降型パーサ

- 手書き
- パーサコンビネータ
- 演算子順位法

下降型手書きパーサ

- 文法要素ごとに関数を1つ書く方法
- 原理的にはわかりやすい
- 文法要素が増えると同じ事何回も書く
のがめんどくさい

パーサコンビネータ

- パーサを組み合わせてパーサを作れる
- 字句解析器が不要
- パーサコンビネータライブラリが必要
- 文法そのものは自由
- 優先順位付き演算子の後付けが不可能

下降型演算子順位法

- 演算子と優先順位の表を用意する
- 優先度を見るパーサを書くだけでOK
- 実行時に優先順位付演算子を追加可能
- S式相当の言語が作れる
- 下降型の演算子順位法を使おう！

SOURCE CODE

parse.scala(lexer, main)

```
package ddc
object parse {
    // 字句解析

    var src = "" // 解析中のソースコード
    var token:Any = "" // トークン
    var ptoken:Any = "" // 1つ前のトークン

    // 正規表現

    val comments = """(?s)^[\t\r\n]*(\#[^\r\n]*)(.*$)""".r
    val nums = """(?s)^[\t\r\n]*([0-9]+)(.*$)""".r
    val ns = """(?s)^[\t\r\n]*([a-zA-Z_][a-zA-Z_0-9]*|[\(\)\{\}\+,=;])(.*$)""".r
    // 字句解析関数

    def lex():Any = {
        ptoken = token
        src match { // ソースを正規表現でマッチさせてtokenに保存
            case comments(a,b) => src = b; lex()
            case nums(a,b) => token = a.toInt; src = b
            case ns(a,b) => token = a; src = b
        }
        ptoken // 前のトークンを返す
    }
    // 予期したトークンを1つ食べる

    def eat(e:Any):Any = {
        if(lex() != e) {
            throw new Exception("syntax error: found unexpected token "+ptoken)
        }
        ptoken
    }
}
```

```
// メインプログラム

def main(argv:Array[String]) {
    val prg = "main=fun() {println(add(1,2,3))} add=fun(a,b,c) return a+b+c"
    val st = parse(prg)
    println("st="+st)
    val ast = st2ast(st)
    println("ast="+ast)
    val s = setmem(ast)
    val e = expand(s)
    val m = memAlloc(e)
    emit("e.s", m)
    exec("gcc -m64 -o e e.s lib.c") match {
        case 0 => exec("./e")
        case _ =>
    }
}
```

parse.scala(parser)

```
// 前置演算子表 演算子=>(優先順位,種類,パラメータ)  
  
def prs(a:Any):Any = a match {  
    case "fun"=>(0,"st")  
    case "{"=>(0,"p","}")  
    case "("=>(0,"p","))"  
    case "return"=>(0,"l")  
    case _=>-1  
}  
  
// 中置演算子表 演算子=>(優先順位,種類,パラメータ)  
  
def ins(a:Any):Any = a match {  
    case "+"=>(10,"l")  
    case "="=>(5,"r")  
    case ","=>(3,"l")  
    case "("=>(0,"p","))"  
    case ";"=>(0,"e")  
    case _=>-1  
}  
  
// パーサ本体  
  
def apply(str:String):Any = {  
    src = str  
    token = ptoken = ""  
    lex()  
    loop(exp(0))  
}  
  
// ループ  
  
def loop(t:Any):Any = token match {  
    case ""=>t  
    case _=>val e=(t,"@",exp(0));loop(e)  
}
```

```
// 式  
  
def exp(p:Int):Any = {  
    if(token == "(" || token == ")") return "void"  
    var t = prs.lex() // 前置演算子  
    in(t) // 中置演算子  
}  
  
// 前置演算子  
  
def pr(t:Any):Any = {  
    val op = t  
    prs(op) match { // 表引いて値を返す  
        case (_, "ep")=>"void"  
        case (np:Int,"st")=>eat("(");val e=exp(np);eat(")");(op,"(",e,")",exp(0))  
        case (np:Int,"p",ep)=>val e=exp(np);(op,e,eat(ep))  
        case (np:Int,"l")=>(op,exp(np))  
        case _=>op  
    }  
}  
  
// 後置演算子  
  
def in(t:Any):Any = {  
    ins(token) match { // 表引いて値返す  
        case (np:Int,"e") if(np >= p)=>val op=lex();in(t,op)  
        case (np:Int,"l") if(np > p)=>val op=lex();in(t,op,exp(np))  
        case (np:Int,"r") if(np >= p)=>val op=lex();in(t,op,exp(np))  
        case (np:Int,"p",ep)=>val sp=lex();val e=exp(np);in(t,sp,e,eat(ep))  
        case _=>t  
    }  
}
```

アジェンダ

＊ マーケティング

＊ インタプリタ

＊ どう新しくする？

＊ マクロ

＊ コンパイルインフラス
トラクチャ

＊ コンパイラ

＊ パーサ(構文解析器)

＊ 今後

インタプリタ

構文木をそのまま実行する

インタプリタの作成

- 構文木を再帰呼び出しして実行するのが最も原始的で簡単だ
- マッチング構文があると簡単に書ける
- スタックマシン化したり、JITしたりすると高速化できる

アジェンダ

＊ マーケティング

＊ インタプリタ

＊ どう新しくする？

＊ マクロ

＊ コンパイルインフラス
トラクチャ

＊ コンパイラ

＊ パーサ(構文解析器)

＊ 今後

マクロ

プログラムでプログラムを操作する

マクロの実装

- マクロの実装は構文木に対するマッチングと置き換えプログラムを実行すること
- マッチング構文があると簡単に書ける
- マッチングしたデータに対してインタプリタを実行して動作させる

SchemeのSyntaxマクロ

- 古典的マクロは変数名がかぶる恐れがある
- 変数名は冠らない清潔な(ハイジニックな)マクロがあるとよい
- いずれ実装したい

アジェンダ

＊ マーケティング

＊ インタプリタ

＊ どう新しくする？

＊ マクロ

＊ コンパイルインフラス
トラクチャ

＊ コンパイラ

＊ パーサ(構文解析器)

＊ 今後

コンパイラ 機械語に翻訳する

コンパイラの課題

- コンパイラのバックエンドの簡単な入門書がないのが実情
- コンパイラはオブジェクト指向より関数型言語の方が得意分野
- 関数型言語で書いた簡単な入門書を書こう

書籍、文書

- ドラゴンブック
- Cで書く古典的
- タイガーブック
- MLで書いてあるけど難しい
- MinCaml
- OCaml美しいが実装がレベル高すぎ
- ふつぱイラ
- Java。実装デ力過ぎ

コンパイラ

- フロントエンド
- 字句解析、構文解析
- バックエンド
- 意味解析、アセンブラー出力

なぜ難しい？

- フロントエンドから順番に解説
してる物が多い
 - 一番最後にアセンブラが書いて
ある
 - アセンブラが凄く難しく感じる
 - 出来上がったソースしかない
- 簡単そう
- フロントエンド
- ソース
- 字句解析
- パーサ
- 意味解析
- コード出力
- アセンブラ
- ↓
- バックエンド
- 難しそう

どうする？

難しそう

- アセンブラと関数型言語を勉強 フロントエンド
しつつバックエンドからーから
作れるといい
- 小さいところから始める
- 今度はフロントエンドが難しく バックエンド
感じる

ソース

字句解析

パーサ

意味解析

コード出力

アセンブラ



橋のようにしよう

フロントエンドとバックエンドは
分けて考えよう

難しそう



小さい橋から作る

小さい橋 簡単

字句解析

パーサ

ソース

インタプリ

安心して

自信をつける



大きい橋 わかってるから簡単

意味解析

パーサ

コード出力

字句解析

ソース

アセンブラ

それでもデカイ？

パスごとに作るようになる

部品を集めて完成させる

テストデータ

出力

みそしる

抽象構文木変換

定数展開

マクロ

バナナ

平たくする

パーサ

さとう

ポテチ

メモリ割り付け

サラダ

生クリーム

字句解析

しょうゆ

コード出力

ソース

いちご

アセンブラ

しょうゆは醤油屋さん、生クリームは牧場で作ればいいよ

サラダは切った奴作っておけば、ならべるだけでいいよ

INDEX

目次

- C言語の簡単なプログラムを書く
- アセンブラーの出力を見る
- Scalaでアセンブラーを出力する
- 便利関数を作る
- Scalaでアセンブラーを実行して動かす

目次2

- データ構造からアセンブラーを出力する
- 変数にメモリ割り付けを行う
- 複雑な式を、シーケンシャルな命令列
に置き換える
- 式の定数を展開する

目次3

- 構文木から、抽象構文木に変換する
- 文字列から構文木を作成する
- 関数単位のコンパイルが出来るようになる
- 条件分岐が出来るようになる

既存文書との違い

- アセンブラ周りから一から作るので
バックエンドの敷居が低く楽しい
- 関数型言語で書いているから簡単
- Scalaだから、忘れにくい
- `x86_64`だから将来も安心

SOURCE CODE

main.scala

```
package ddc
import java.io._
object main {
  def main(argv:Array[String]) {
    val prg = exec.readAll(new FileInputStream(argv(0)))
    val st = parse(prg)
    val ast = st2ast(st)
    val s = setmem(ast)
    val e = expand(s)
    val m = memAlloc(e)
    emit("e.s", m)
    exec("gcc -m64 -o e e.s ddc/lib.c") match {
      case 0 => exec("./e")
      case _ =>
    }
  }
}
```

```
# tes.dd
main=fun() {
  printInt(add(1,2,3))
}

add=fun(a,b,c) return a+b+c
```

```
>scala ddc.main tes.dd
>/tes
6
```

asm.scala exec.scala

genid.scala

```
// ファイルを開いて、アセンブラを出力する
// asm関数
package ddc

import java.io._
object asm {
  var p:PrintWriter = null
  def open(file:String) {
    p = new PrintWriter(new BufferedWriter(new FileWriter(file)))
  }
  def apply(s:String) {
    p.println(s)
  }
  def close() {
    p.close()
  }
}

// 使い方
def main(argv:Array[String]) {
  asm.open("a.txt")
  asm("test")
  asm.close()
}
```

```
// ユニークなIDを生成
// genid関数
package ddc;
object genid {
  var counter = 0
  def apply(s:String):String = {
    counter += 1
    s + counter
  }
}
```

```
// プロセス実行して、出力し、リターン値を返す
// exec関数
package ddc
import java.io._
object exec {
  def apply(cmd:String):Int = {
    val p = Runtime.getRuntime().exec(cmd)
    print(readAll(p.getInputStream()))
    print(readAll(p.getErrorStream()))
    p.waitFor()
  }
  def readAll(p:InputStream):String = {
    def f(s:String, i:BufferedReader):String = {
      i.readLine() match {
        case null => s
        case a => f(s+a+"\n", i)
      }
    }
    f("",new BufferedReader(new InputStreamReader(p)))
  }
}

// 使い方
def main(argv:Array[String]) {
  exec("ls")
}
```

emit.scala

```
// コード出力関数
package ddc;
object emit {

  def apply(filename:String, ls>List[Any]) {
    asm.open(filename)
    ls.foreach {
      case (name:String, body>List[Any]) =>
        asm(".globl "+name)
        asm(name+":")
        asm("\tpushq\t%rbp")
        asm("\tmoveq\t%rsp, %rbp")
        body.foreach {
          case ("movl", a, b) => asm("movl "+a+", "+b)
          case ("subq", a, b) => asm("subq "+a+", "+b)
          case ("addl", a, b, c) =>
            asm("movl "+a+", %eax")
            asm("addl "+b+", %eax")
            asm("movl %eax, "+c)
          case ("call", n, b>List[Any]) => prms(b, regs); asm("call "+n)
          case ("ret", a) =>
            asm("movl "+a+", %eax")
            asm("leave")
            asm("ret")
        }
        asm("\tleave")
        asm("\tret")
    }
    asm.close()
  }
}
```

```
val regs = List("%edi", "%esi", "%edx")
def prms(ps>List[Any], rs>List[Any]) {
  (ps, rs) match {
    case (List(), _) =>
    case (p::ps, r::rs) =>
      asm("movl "+p+", "+r)
      prms(ps, rs)
  }
}

// 使い方
def main(argv:Array[String]) {
  emit("emit.s", List(
    ("_main", List(
      ("movl", "$1", "%edi"),
      ("call", "_printInt", List())
    ))
  ))
  exec("gcc -m64 -o emit emit.s lib.c") match {
    case 0 => exec("./emit")
    case _ =>
  }
}
```

memAlloc.scala

```
// 変数 aをスタック上のメモリに割当てる

package ddc
object memAlloc {
    var m:Map[String,String] = null
    def apply(ls>List[Any]):List[Any] = ls.map {
        case (n:String,ls>List[Any])=>
            counter = 0
            m = Map()
            val ll = ls.map(g)
            val size = ((15-counter)/16)*16
            (n,("subq",$"+size,"%rsp")::ll)
    }

    def g(l:Any):Any = l match {
        case ("movl", a, b) => ("movl", adr(a), adr(b))
        case ("addl", a, b, c) => ("addl", adr(a), adr(b),adr(c))
        case ("call", a, b>List[Any]) => ("call", a, b.map(adr))
        case ("ret", a) => ("ret", adr(a))
    }

    var counter = 0
    def adr(a:Any):Any = a match {
        case a:String if(m.contains(a))=> m(a)
        case a:String if(a.substring(0,1)=="%" || a.substring(0,1)=="$") => a
        case a:String => counter -= 4; val n = counter + "(%rbp)"; m = m + (a -> n); n
        case a => a
    }
}
```

```
// 使い方

def main(argv:Array[String]) {
    val prgs = List(
        ("_main",List(
            ("movl", "$l", "a"),
            ("call", "_printInt",List("a"))
        ))
    )
    val l = memAlloc(prgs)
    println("l=" + l)
    emit("m.s",l)
    exec("gcc -m64 -o m m.s lib.c") match {
        case 0 => exec("./m")
        case _ =>
    }
}
```

expand.scala

// 複雑な式を展開して平たくする

```
package ddc
object expand {

  def argv(as>List[String], rs>List[Any]):List[Any] = (as, rs) match {
    case (List(), rs) => List()
    case (a::as, r::rs) => ("movl", r, a)::argv(as, rs)
  }

  val regs = List("%edi", "%esi", "%edx", "%ecx", "%r8d", "%r9d")
  def f(l>List[Any], e:Any):(List[Any], String) = e match {
    case ("add", a, b) =>
      val id = genid("ex_")
      val (la, al) = f(l, a)
      val (lb, bl) = f(la, b)
      ("addl", al, bl, id)::bl,id)
    case ("mov", a:String, id:String) => (("movl", a, id)::l, id)
    case ("mov", a:Int, id:String) => (("movl", ("$"+a), id)::l, id)
    case ("mov", a, id:String) =>
      val (l2, idl) = f(l, a);(("movl", idl, id)::l2, id)
    case ("call", a, b>List[Any]) =>
      var (la,ids) = b.foldLeft((l, List[String]())){
        case ((l,ids),b)=> val (l2,id) = f(l, b); (l2,id::ids)
      }
      ("call", a, ids)::la, "%eax")
    case ("ret", e) => val (l2, id) = f(l, e);(("ret", id)::l2, id)
    case id:String => (l, id)
    case e => (e::l, null)
  }
}
```

// 本体

```
def apply(p>List[Any]):List[Any] = p.map {
  case (n,a>List[String],b>List[Any]) =>
    val ll = b.foldLeft(argv(a,regs)){
      case (l,b)=> val (l2, id) = f(l, b); l2
    }
    (n,ll.reverse)
}
// 使い方
def main(argv:Array[String]) {
  val prg = List(
    ("_main", List(), List(
      ("mov", 10, "a"),
      ("mov", 1, "b"),
      ("mov", 2, "c"),
      ("mov", ("call", "_println", List(("add", "a", ("add", "b", "c")))), "d"),
      ("ret", "d")
    )),
    ("_add", List("a", "b"), List(
      ("ret", ("add", "a", "b"))
    ))
  )
  val p = expand(prg)
  println("p=" + p)
}
```

setmem.scala

```
// 定数をメモリに展開する

package ddc
object setmem {
  var ls:List[Any] = List()

  def apply(e:List[Any]):List[Any] = e.map {
    case (n:String, a:List[String], b:List[Any]) =>
      ls = List()
      val b2 = b.map(f)
      (n,a,ls:::b2)
  }
  def f(e:Any):Any = e match {
    case ("mov", a, b) => ("mov", f(a), f(b))
    case ("ret", a) => ("ret", a)
    case ("add", a, b) => ("add", f(a), f(b))
    case ("call", a, b:List[Any]) => ("call", a, b.map(f))
    case a:Int =>
      val id = genid("s_")
      ls = ("mov",a,id)::ls
      id
    case a => a
  }
}
```

```
// 使い方

def main(argv:Array[String]) {
  val prg = List(
    ("_main",List(),List(
      ("call","_println",List(("call","_add",List(1,2,30))))
    )), 
    ("_add", List("a","b","c"),List(
      ("ret",("add","a",("add","b","c")))
    ))
  )
  val s = setmem(prg)
  println("s="+s)
  val e = expand(s)
  val m = memAlloc(e)
  emit("m.s", m)
  exec("gcc -m64 -o m m.s lib.c") match {
    case 0 => exec("./m")
    case _ =>
  }
}
```

st2ast.scala

```
package ddc
object st2ast {

  def f(fn:Any):Any = fn match {
    case (n,"=",("fun",("","a,""),b)) => ("_"++n, params(a), bodys(b))
  }

  def params(e:Any):List[Any] = e match {
    case (a,",",b) => params(a):::params(b)
    case "void"=>List()
    case a => List(a)
  }

  def fargs(e:Any):List[Any] = e match {
    case (a,",",b) => fargs(a):::fargs(b)
    case a => List(exp(a))
  }

  def exp(e:Any):Any = e match {
    case ("{",b,"}") => bodys(b)
    case ("(",b,")") => exp(b)
    case (a,"(",b,")") => ("call", "_" + a, fargs(b))
    case (a,"=",b) => ("mov", exp(b), exp(a))
    case (a,"+",b) => ("add", exp(a), exp(b))
    case ("return", a) => ("ret", exp(a))
    case (a,";") => exp(a)
    case a:Int => a
    case a:String => a
  }
}
```

```
def bodys(e:Any):List[Any] = e match {
  case (a,"@",b) => bodys(a):::bodys(b)
  case a =>
    exp(a) match {
      case e>List[Any] => e
      case a => List(a)
    }
}

def apply(st:Any):List[Any] = st match {
  case (a,"@",b) => f(a):::List(f(b))
}

//使い方

def main(argv:Array[String]) {
  val st =
    (("main","=",("fun",("void",""),
      ("{",("println",("(("add",("(",(1,"",(2,"",3)),")"),")"),")"),
      "}")),
    "@",
    ("add","=",("fun",("(","a","","("b","","c"),")"),
      ("a","+",("b","","c")))))
  )
  val ast = st2ast(st)
  println("ast=" + ast)
}
```

簡単でしょ？

- Cyber X に来て数ヶ月で作りました
- 基本的なバックエンドの構造を簡単に表す事が出来ていると思います
- これなら作れそうな気がしませんか？

アジェンダ

- ＊ マーケティング
- ＊ インタプリタ
- ＊ どう新しくする？
- ＊ マクロ
- ＊ コンパイルインフレス
- ＊ コンパイラ
- ＊ トラクチャ
- ＊ 今後
- ＊ パーサ(構文解析器)

今後 この先の目標と予定

今後の予定

- ✿ 基本的で本質的な理解は出来た
- ✿ 文書としてまとめているところ
- ✿ 実用的な実装を作る
- ✿ 実際にアプリを作る
- ✿ ライブドアを作成する

予定

- ✿ 具体的な仕様を策定する
- ✿ リファレンスマニュアル等を充実させる
- ✿ エラー処理を組み込む
- ✿ 最適化をする
- ✿ 型、配列、構造体、クラス等の導入

予定

- ✿ 複数CPUに対応する
- ✿ 他の言語も作る
- ✿ CGIを作る
- ✿ オブジェクト指向化する
- ✿ きりがないです