Scalaで作る x86_64コンパイラ

h_sakurai

今日のはなし

- 最も簡単なコンパイラ
- x86_64アセンブラ
- アセンブラを拡張した物がコンパイラ

とつぜんですが、

最も簡単なコンパイラを

作りました。

e.c e.s e.scala

```
/* e.c */
int add(int a, int b) {
  return a + b;
}

void main() {
  int a = add(3, 4);
  printf(“%d¥n”, a);
}
```

```
; e.s

        .text
.globl _add
_add:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        movl    -8(%rbp), %eax
        addl    -4(%rbp), %eax
        leave
        ret
        .cstring
LC0:
        .ascii "%d\12\0"
        .text
.globl _main
_main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    $4, %esi
        movl    $3, %edi
        call    _add
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %esi
        leaq    LC0(%rip), %rdi
        movl    $0, %eax
        call    _printf
        leave
        ret
```

```
// e.scala
package ccc
object add {
  def main(argv:Array[String]) {
    asm.open(“e.s”)
    asm("""
        .text
.globl _add
_add:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        movl    -8(%rbp), %eax
        addl    -4(%rbp), %eax
        leave
        ret
        .cstring
LC0:
        .ascii "%d\12\0"
        .text
.globl _main
_main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    $4, %esi
        movl    $3, %edi
        call    _add
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %esi
        leaq    LC0(%rip), %rdi
        movl    $0, %eax
        call    _printf
        leave
        ret
    """)
    asm.close()
    exec(“gcc e.s -o e”)
  }
}
```

```
$ gcc e.c
$ ./a.out
7
$ gcc -S e.s
$ vi e.s
$ cp e.s e.scala
$ vi e.scala
$ scalac e.scala
$ scala ccc.e
$ ./e
7
```

アセンブラを出力し

てgccよぶだけなら

簡単ですよね

x86_64アセンブラ

- x86を64bitに拡張したもの

gccのアセンブラ

- gccのアセンブラは基本的に右から左へ値を入れます。
  - mov $1, %eaxで1を%eaxに入れます
  - コンパイラ作るときはgcc使うと楽なので覚えるといいです。

ディレクティブ

- "."から始まるアセンブラに対する命令
- .cstring: テキストデータを含むセクションの開始
- .ascii: 文字列定数
- .text: プログラムコードの開始
- .globl: ラベル名をグローバルに公開する

ラベル

- アドレスに名前を付けた物をラベルと呼びます。識別子 ":" と記述します。
- 例) abc:

インストラクション

- pushq, popq 64bitのスタック操作
- movq 64bitの転送命令
- subq 64bitの引き算
- movl 32bitの転送命令
- leaq アドレスをレジスタに入れる

インストラクション

- addl 32bitの足し算
- call 関数の呼び出し
- leave 関数の終了処理
- ret 関数から呼び出し元に戻る

レジスタ

- CPU内の記憶領域
- %eax, %ebx, %ecx, %edx, %edi, %esi, %rbp, %r8d, %r9d, %rsp
- %rspがスタックポインタ
- %rbpが関数フレームの先頭を表す

インデックス参照

- -4(%rbp)等と記述
- %rbpから-4のアドレスの中身を表す

即値

- $から始まる数値
- 例) $1

関数呼び出し

```
movl    $4, %esi
movl    $3, %edi
call    _add
movl    %eax, -4(%rbp)
```

- 引数はレジスタに格納
- レジスタに格納出来なければスタックに格納
- call で呼び出す
- 戻り値は%eaxに入ってくる

関数の実装

- ベースポインタをスタックに
- スタックポインタをベースポインタに
- レジスタから引数を取得してメモリに保存
- 関数の処理を実行
- eaxに結果を保存
- leaveでpushq,movqを戻す
- retで呼び出し元に戻る

```
_add:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        movl    -8(%rbp), %eax
        addl    -4(%rbp), %eax
        leave
        ret
```

このような知識で

コンパイラは作れます

アセンブラを拡張してコンパイラにしてみる

- Scalaでアセンブラを拡張して行って最終的にコンパイラを作ります
- いつも実行ファイルが動くので楽しい
- Anyと多値やListを使い記述
- Scalaのmatch使って短く書く

補助関数

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

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("e.s")
    asm("test")
    asm.close()
  }
}
```

```
// ユニークなIDを生成
// genid関数
package ccc
object genid {
  var counter = 0
  def apply(s:String):String = {
    counter += 1
    s + counter
  }
  // 使い方
  def main(argv:Array[String]) {
    println(genid("a"))
    println(genid("a"))
    println(genid("a"))
  }
}
```

```
// プロセス実行して、出力し、リターン値を返す
// exec関数
package ccc
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")
  }
}
```

データをアセンブル

```
// データからアセンブラを出力できる

package ccc
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("\tmovq\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("e.s", List(
    ("_main",List(
      ("movl", "$1", "%edi"),
      ("call", "_printInt",List())
    ))
  ))
  exec("gcc -m64 -o e e.s ccc/lib.c")
}

}
```

変数を導入

```
// 変数が使えるようにする

package ccc
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", "$1", "a"),
      ("call", "_printInt",List("a"))
    ))
  )
  val l = memAlloc(prgs)
  println("l="+l)
  emit("e.s",l)
  exec("gcc -m64 -o e e.s ccc/lib.c")
  }

}
```

ネストした式の導入

```
// ネストした式が使える
package ccc
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, a1) = f(l, a)
      val (lb, b1) = f(la, b)
      (("addl", a1, b1, id)::lb,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, id1) = f(l, a); (("movl", id1, 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", "_printInt", 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)
  }
}
```

定数の導入

```
// 定数が使える

package ccc
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","_printInt",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("e.s", m)
  exec("gcc -m64 -o e e.s ccc/lib.c")
 }
}
```

C風構文の導入

```
// C風の構文が使えるようにする

package ccc
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(a)
    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",")",
      ("{",("printInt","(",("add","(",(1,",",(2,",",3)),")"),")"),")"),"}")
    )),
    "@",
    ("add","=",("fun","(",("a",",",("b",",","c")),")",
      ("a","+",("b","+","c")))
    ))
    val ast = st2ast(st)
    println("ast="+ast)
  }
}
```

パーサの導入(レキサ)

```
// テキストから読み込めるようにする
package ccc
object parse {
  // メインプログラム
  def main(argv:Array[String]) {
    val prg = "main=fun() {printInt(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 ccc/lib.c")
  }
```

```
// 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
  }
```

パーサの導入(パーサ)

```
// 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 = pr(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
  }
}
}
```

ファイル読込の導入

```
// ファイルから読み込めるようにする

package ccc
import java.io._
object main {
  def main(argv:Array[String]) {
    val src = exec.readAll(new FileInputStream(argv(0))) // ファイルから読み込む
    val st = parse(src) // 7. テキストデータを使えるように
    val ast = st2ast(st) // 6. Cっぽい構文を使えるように
    val s = setmem(ast) // 5. 定数を使えるように
    val e = expand(s) // 4. ネストした式が使えるように
    val m = memAlloc(e) // 3. 変数が使えるようにする
    emit("e.s", m) // 2. データからアセンブラを出力する
    exec("gcc -m64 -o e e.s ccc/lib.c") // 1. gccでコンパイルできる
  }
}
```

```
# e.ccc
main=fun() {
  printInt(add(1,2,3))
}
add=fun(a,b,c) return a+b+c
```

```
>scala ccc.main e.ccc

>./e
6
```

コンパイラ

完成！！

コンテンツ

- github
  - http://github.com/hsk/x86_64
- アセンブラを学びながらScalaで作るコンパイラバックエンド入門
  - http://hsk.github.com/timi/
- ドキュメント
  - http://hsk.github.com/x86_64/x86_64.pdf
  - http://hsk.github.com/x86_64/x86_64.ppt
  - http://hsk.github.com/x86_64/x86_64.key

Demo

まとめ

- アセンブラから始めればネイティブコンパイラも簡単
- 徐々に拡張していくと常に動くものが出来て楽しく自信を持って開発出来る
- Scalaなら分かりやすく思い出しやすい実装が作れる