YARVの ソースを 読んでみた (コンパイラ前段編)

http://d.hatena.ne.jp/hzkr/19000101

の

まとめ

2006/12/22 (Fri) / d:id:hzkr

YARV とは

- プログラミング言語 Ruby の処理系
  - のひとつ
  - ささだこういち さんによる実装
  - 本家にマージされるらしい (1.9.1?)
- 特徴
  - Rubyコードを仮想マシン語にコンパイルして実行
  - 速い！
- URI
  - http://www.atdot.net/yarv/
  - http://www.atdot.net/svn/yarv/trunk/ ソース
  - rev. 584 を読んでいます

参考資料

- YARV Maniacs
  - http://jp.rubyist.net/magazine/?0006-YarvManiacs
- Ruby ソースコード完全解説
  - http://i.loveruby.net/ja/rhg/
- Ruby リファレンスマニュアル
  - http://www.ruby-lang.org/ja/man/

流れ（mainからyarvまで）

- main @ main.c
  - ruby_init @ eval.c
    - 組み込みモジュールの初期化など
  - ruby_options @ eval.c
    - この辺りで構文解析。本家Rubyと共通。
      (Rubyのコード文字列を、NODE型の木構造に変換)
  - ruby_run @ eval.c
    - ruby_exec @ eval.c
      - ruby_exec_internal @ eval.c
        - yarvcore_eval_parsed @ yarvcore.c

ここから
YARV 評価器
スタート

流れ（yarv評価器内）

- yarvcore_eval_parsed @ yarvcore.c
  - th_compile_from_node @ yarvcore.c
    - 構文木を、YARVマシン語列に変換（コンパイル）
    - yarv_iseq_new_with_opt @ iseq.c
      - iseq_compile @ compile.c
        - iseq_compile_each @ compile.c
          構文木→マシン語列の変換関数
        - iseq_setup @ compile.c
          最適化などなど
  - yarvcore_eval_iseq @ yarvcore.c
    - マシン語列を、実行

とりあえず
ここまで
読んだ

このスライドの、この後の流れ

- iseq_compile_each の雰囲気
  - ふんいき
- 使われてるマクロ
  - これだけ覚えとけば読める!! はず!!!
- ダイジェスト版 iseq_compile_each
  - 詳しくはソースの実物を読んでね☆

とりあえず
ここまで
読んだ

iseq_compile_each @ compile.c

- 巨大なswitch文
  - ひとつひとつの case が、
    Rubyの構文ひとつひとつに対応

```
static int
iseq_compile_each(...)
{
  ...
  switch()
  {
  case NODE_IF: ...          // if～then～else～end
  case NODE_LASGN: ...       // ローカル変数への代入
  ...                        // などなど...
  }
}
```

iseq_compile_each @ compile.c

- それぞれのcaseで、マシン語生成（例）

```
LINK_ANCHOR* ret;
...
case NODE_NOT:{   // “!foo” のような式のコンパイル
   COMPILE(ret, “value”, node->nd_body); //式fooをコンパイル
   ADD_INSN(ret, nd_line(node), putnot);   //putnot命令を生成
   if (poped) {
      ADD_INSN(ret, nd_line(node), pop);     // pop命令を生成
   }
}
```

- マクロ色々
  - COMPILE_xxx
  - ADD_xxx

マクロ色々

- COMPILE_
- COMPILE_POPED
- COMPILE
  - iseq_compile_each の再帰呼び出し。
    上から順に、第5引数poppedを
    - falseに固定
    - trueに固定
    - 自分で指定
  - 式の値が必要な場合（1+(2+3) の 2+3 など）は
    popped = false で、不要な場合（メソッドの最後じゃない文全体など）は true で呼び出す。
    - 本当はいらない場面で無駄に式の値をメモリに載せたり
      おろしたりしないように。
    - 式を評価するときの副作用が重要なので、値が不要だからって
      実行やコンパイルが全部省略されるわけではないです

マクロ色々

- LIST_ANCHOR (これはマクロじゃない)
  - YARV命令列を表現する構造体
- DECL_ANCHOR(v)
  - 新しい LIST_ANCHOR v を宣言
- ADD_INSN
- ADD_INSN1
- ADD_INSN2
- ADD_INSN3 ...
  - LIST_ANCHORの末尾に1個命令を追加
- ADD_LABEL
  - LIST_ANCHORの末尾に1個ラベルを追加
    （ラベル：ifなどジャンプ命令の飛び先指定に使う）
- ADD_SEQ
  - リスト2つを結合

ダイジェスト版 iseq_compile_each

- NODE_IF
  - こんなふんいきです
- ループとイテレータと例外と部屋とYシャツと私
  - と break と ensure
- メソッド呼び出し
  - Rubyはなんでもかんでもメソッド呼び出しなので、“+演算子のコンパイル!” とかみたいに個別に考える必要はなくて、↑や↓に上げた特別な構文以外はだいたい全部メソッド呼び出しなのでした
- だいにゅー
  - たじゅー
- 定義文
  - class, def, module, …

NODE_IF

```
if nd_cond then
  nd_body
else
  nd_else
end
```

↓ ↓ こうコンパイル↓ ↓

```
  (nd_condをコンパイルしたコード)
  branchunless else_label
then_label:
  (nd_bodyをコンパイルしたコード)
  jump end_label
else_label:
  (nd_elseをコンパイルしたコード)
end_label:
```

※ この部分は
実際はもっと賢い
(Yarv Maniacs 第7回)

NODE_IF のソース

```
case NODE_IF:{
  DECL_ANCHOR(cond_seq);
  DECL_ANCHOR(then_seq);
  DECL_ANCHOR(else_seq);
  LABEL *then_label, *else_label, *end_label;
  then_label = NEW_LABEL(nd_line(node));
  else_label = NEW_LABEL(nd_line(node));
  end_label = NEW_LABEL(nd_line(node));

  compile_branch_condition(iseq, cond_seq, node->nd_cond, then_label, else_label);
  COMPILE_(then_seq, "then", node->nd_body, poped);
  COMPILE_(else_seq, "else", node->nd_else, poped);
  ADD_SEQ(ret, cond_seq);
  ADD_LABEL(ret, then_label);
  ADD_SEQ(ret, then_seq);
  ADD_INSNL(ret, nd_line(node), jump, end_label);
  ADD_LABEL(ret, else_label);
  ADD_SEQ(ret, else_seq);
  ADD_LABEL(ret, end_label);
  break;
}
```

変数宣言

賢く分岐するコードを生成する関数

nd_bodyとnd_elseをコンパイル

全部つなげる

こんな調子です

ループとイテレータと例外

- ループ
  - while
- イテレータ
  - for, ブロック付きメソッド呼び出し, yield
- ループ制御
  - break, next, redo
- 例外
  - begin～rescue～ensure～end, raise

ループとイテレータと例外

- イテレータからのbreak等による脱出は内部的には例外の仕組みで実装されてる
- whileループでも、スコープをまたがるbreak等は例外の仕組みで実装されている

```
while … do
  class X; break; end
end
```

- break等でbegin～ensureを抜けるとき、ちゃんとensure節を実行しないとダメ

ループとイテレータと例外

- …というわけで、
  このへんは(ちょっと)複雑にからみあってます。
- コンパイル処理中に管理される情報
  iseq->compile_data
  - start_label
  - redo_label
  - end_label
    - 「今 next, redo, break 文が出たらどこに飛ぶジャンプ命令に変換すべきか？」
    - nil なら例外を発生させる命令に変換されます
  - ensure_node_stack
    - 「今 next, redo, break 文で飛び出す場合に、実行しなきゃ行けない ensure 節の全リスト（スタック）」

ensure

- add_ensure_iseq @ compile.c
  - 同じスコープ内のensure節の中身は、break文のある場所に逐一展開される実装

```
while true
  begin
    break
    break
  ensure
    print “hello”
  end
end
```

```
0000 putself
0001 putstring      "hello"
0003 send           :print, 1, nil, 4, <ic>
0009 pop
0010 putnil
0011 leave
0012 putself
0013 putstring      "hello"
0015 send           :print, 1, nil, 4, <ic>
0021 pop
0022 putnil
0023 leave
0024 putself
0025 putstring      "hello"
0027 send           :print, 1, nil, 4, <ic>
0033 pop
0034 jump           0
0036 putnil
0037 leave
```

  - なので、ensureから例外が飛ぶとき二重処理しない工夫がある http://d.hatena.ne.jp/hzkr/20061124#p11
    - どのハンドラがどの例外を捕まえるかの対応関係を、単純なコード上の一区間でなく、もっと細分化している

メソッド呼び出し

- super と yield も似たような感じです

```
obj.method( arg1, arg2, arg3 ) { block }
```

- 流れ
  - blockは別の命令列オブジェクトとして コンパイルしておく
  - objを評価するコードを生成
  - 引数を左から右に評価するコードを生成
  - 呼び出し命令を生成
    - メソッド : YARV の send 命令を生成
    - super : invokesuper 命令を生成
    - yield : invokeblock 命令を生成
- 以上

代入

- 変数・定数の種類
  - methodlocal =
  - blocklocal =
  - @instancevar =
  - @@classvar =
  - $globalvar =
  - ConstantVar =
- に応じて、それぞれYARV命令があるので
  - setlocal, setdynamic, setinstancevariable, setclassvariable, setglobal, setconstant, ...
- それにコンパイルするだけ

代入のなかま:属性参照

- 属性参照

```
obj.attr = 100
```

- は、メソッド呼び出し

```
obj.send( :attr=, 100 )
```

- と同じと思っていませんでしたか？

```
obj = nil
class <<obj
   def attr(x): 200: end
end
p (obj.attr = 100)          # 100
p (obj.send(:attr=, 100))   # 200
```

- 式の値を右辺の値にする工夫あり（普通のメソッド呼び出しに加えて、計算スタックの値を途中で書き換え）

代入のなかま:多重代入

- こんな感じの一気に代入する文

```
a1, a2, a3, a4, a5 = b1, b2, b3, *bb
```

- a1=b1; a2=b2; … とは違います

```
x, y = y, x        # xとyの値を入れ替え
                   # x=y; y=x だと両方yになってしまう
```

- 「まず右辺を左から右に評価して、
  どんどん値をスタックにpush」
  →「次に、値をpopしながら、
  左辺の変数へ右から左の順番で代入」
  （スタックなので逆順）という実装でした

クラス定義、メソッド定義

- JavaやC++などと違って
  Rubyではクラス定義やメソッド定義も、
  実行時に実行される「単なる実行文」

- なのでコンパイラ的には、定義文のコンパイルは楽（定義命令に変換するだけ）
  - definemethod : メソッド, 特異メソッド
  - defineclass: クラス, 特異クラス, モジュール

まとめ

- YARVの、構文木→YARV命令列 変換
  - ...の部分のコードを読んだ結果をまとめました
  - 超ダイジェスト版なので、物足りない方はぜひぜひ
    http://www.atdot.net/svn/yarv/trunk/
    を読みましょう!!

とりあえず
ここまで
読んだ

- 続く...
  - 最適化などなど
  - YARV の VM の実装