情報科学 2006　久野クラス　＃8

久野　靖*

2006.12.15

はじめに

前回の抽象構文木は分かりやすかったという意見が多かったようで、よかったです。前回は打ち出しただけだったので「何に使うのか」という質問がありましたが、今回はその答えとして「抽象構文木の解釈実行」「抽象構文木からアセンブリ言語/機械語への変換」を取り上げます。これらを学ぶと、普段お世話になっている Java の処理系の構造も分かって来ると思います。このほか、「例外処理機構」についても (ようやく) 説明します。これで「おまじない」がかなり解消することになると思います。演習問題解説は課題の期限が 2 週間になったので、お楽しみを無くさない程度 (?) にやります。

1　演習問題解説

1.1　演習 1

(a) は前回資料でオートマトンへの変換をやったのでそのオートマトンの受理する言語と同じ。つまり「ab」「abab」「ababab」等、ab が 1 個以上並んだもの。

(b) は $S \Rightarrow aSa \Rightarrow abSba \Rightarrow abcScba \Rightarrow abccba$ 等、加わる記号は a、b、c のどれでも自由だが、必ず S の前後に同じものが増えて行くことに注意。したがって、abc の任意の列 (長さ 0 以上) があり、続いてその列を前後反転したものがくっつく。

(c) は $S \Rightarrow aSBC \Rightarrow aaSBCBC \Rightarrow \cdots$ により、まず a が N 個、BC が N 個の列ができる。次に、B と C が小文字にならなければならないが、B を小文字にするためには左隣に a か b がなければならないので、…abbbb…b という形ができるしかない。同様に C を小文字にするためには左隣に b か c がなければならないので、結局…bccc…c という形ができるしかない。こうなるためには CB → BC を使って全部の C を右に集めるしかない。だから最後にできるものは a が N 個、b が N 個、c が N 個並んだもの ($N > 0$)。

1.2　演習 2

図 1 の通り。(c) は問題の誤植で済みません (元のままだと「導出できない」状態でした)。

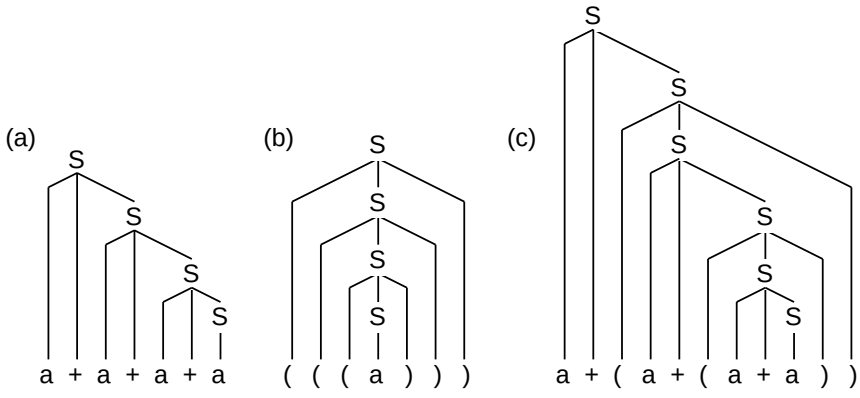

図 1: 演習 2 の構文木

*筑波大学大学院経営システム科学専攻

1.3 演習 3

(a) は前回資料に掲載。(b) と (c) は図 2 の通り。

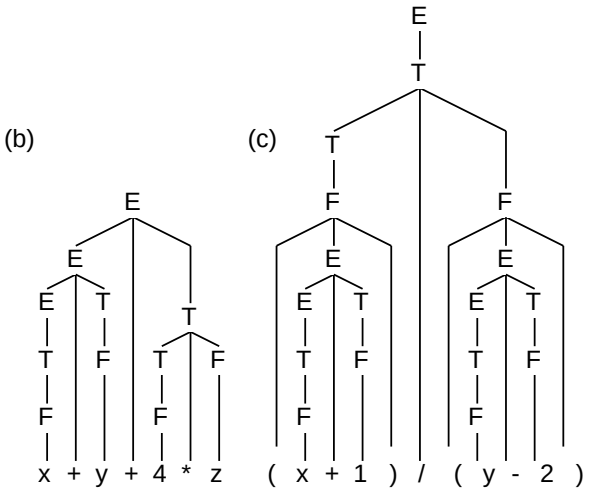

図 2: 演習 3 の構文木

1.4 演習 4

Tree を作るところだけ示す。

```
(a) new Tree('+',
      new Tree('*',
        new Tree('*', new Tree('v',"x"), new Tree('v',"y")),
        new Tree('v',"z")),
      new Tree('i', "3"));

(b) new Tree('/',
      new Tree('+', new Tree('v',"x"), new Tree('i',"1")),
      new Tree('*',
        new Tree('+', new Tree('v',"y"), new Tree('i',"2")),
        new Tree('+', new Tree('v',"z"), new Tree('i',"3"))));

(c) new Tree('-', new Tree('v',"e"),
      new Tree('-', new Tree('v',"d"),
        new Tree('-', new Tree('v',"c"),
          new Tree('-', new Tree('v',"b"), new Tree('v', "a")))));
```

1.5 演習 5

メソッド toString() だけ取り換えればよい。まず前置記法:

```
    public String toStringPrefix() {
      if(op == 'i' || op == 'v') { return (String)left; }
      String l = ""; if(left != null) l = " " + ((Tree)left).toString();
      String r = ""; if(right != null) r = " " + ((Tree)right).toString();
//    return "(" + op + l + r + ")";  // かっこあり
      return op + l + r;              // かっこなし
    }
```

次に後置記法:

```
    public String toString() {
      if(op == 'i' || op == 'v') { return (String)left; }
      String l = ""; if(left != null) l = ((Tree)left).toString() + " ";
      String r = ""; if(right != null) r = ((Tree)right).toString() + " ";
//    return "(" + l + r + op + ")";  // かっこあり
      return l + r + op;              // かっこなし
    }
```

ほとんど変えるところはないというか、空白がうまく空くようにちょっと工夫してるだけですね。

2　抽象構文木から実行へ (1)

2.1　言語処理系の枠組み

非常に遅まきながら、プログラミング言語処理系 (たとえば皆様が使っている Java の処理系など) はどのような構造になっているのか、ということを説明しよう。処理系は皆様がプログラミング言語による記述をおこなったもの (**ソースコード**) を受け取り、最終的にはその記述に対応する動作が実行されるようにする。

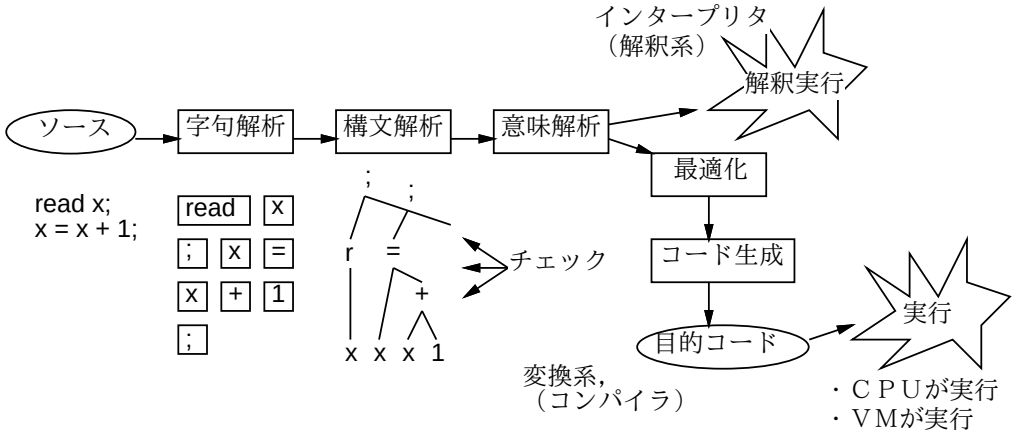

図 3: 言語処理系のフェーズ

処理系の中はいくつかのフェーズに分かれているのが普通である (図 3)。次の 3 フェーズは普通、どの処理系でも共通である:

- **字句解析** (lexical analysis) — コメントなどを削除し、記号 (演算子)、名前、定数などの形式をチェックしてそれぞれのかたまり (字句ないし**トークン**) にまとめる。有限オートマトンや正則 (正規) 表現を用いて形式を記述したものからこの部分の機能を生成するツールが多数あるが、簡単なものなら自分で記述しても大変ではない。これは次回取り上げる。
- **構文解析** (syntax analysis) — 前回やったように、トークンの並びに対して、構文規則に合っているかどうかをチェックし、構文木ないし抽象構文木を内部で組み立てる (少なくとも多くの処理系では)。この処理方法は色々あるが、次回に一例を取り上げる。

  なお、字句の規則も文脈自由文法で表せるので (Type 2 文法は Type 3 文法を包含することに注意)、字句解析の部分も含めて文法として記述し構文解析で処理することも可能ではある。しかしそれでは文法が長く繁雑になり、また処理も遅くなるため、普通はそういうことはしない。
- **意味解析** (semantic analysis) — 構文が正しく、抽象構文木がちゃんとできたら、それでプログラムが正しいというわけには行かない。たとえ構文が正しくても、変数が定義していないとか、型が間違っているとかの誤りは存在する。これらの誤りをチェックするのが意味解析である。

  多くの処理系では、構文解析に失敗した場合は、意味解析に進まない。このため、皆様が Java コンパイラを使っているとき、ある種のエラーメッセージは構文が全部正しくなってから後で始めて表示されることに気がついたと思う。それはそういうわけだったのだ。なお、今回取り上げている簡単な言語は、構文が正しければプログラムとしても正しく動く程度のものなので、意味解析については扱わない。

意味解析が終わったら、その後の道筋は処理系のタイプによって 2 種類に分かれる。

- **解釈実行系** (interpreter) — そのプログラムの意味する動作を直接実行する。
- **変換系** (translator) — そのプログラムの意味する動作を行うような記述を生成する。

解釈実行系では、意味解析が終わったらすぐ実行に進んで終わりである。一方、変換系では、次のような段階を経て出力コードを生成する。

- **最適化** (optimization) — 生成されるコードが効率のよいものであるように、コードを内部的に分析/変形して出力に備える。

- **コード生成** (code generation) — 出力コードを生成する。

変換系のなかでも、CPU が直接実行できるような命令列 (**機械語**) やそれに近いものを出力コードとするようなものを**コンパイラ** (compiler) と呼ぶ。

Java の場合は、CPU が直接命令を実行するのではなく、機械語と同じぐらいの水準の命令を解釈実行するようなプログラム (**仮想機械**ないし **VM** — virtual machine) が各種環境ごとに用意されていて、このプログラムが「バイトコード」と呼ばれる Java 用の仮想機械語を実行する。このため、`javac` でコンパイルしたコード (`.class` ファイル) は、さまざまな環境に持って行って実行できる。そして `java` コマンドは実は VM プログラムを実行するコマンドだったわけだ。

ところで、機械語は直接人間が読むのには大変なので、実際には**アセンブリ言語** (assembly language) と呼ばれる、命令や番地やレジスタなどを名前で表す書き方が多く使われる。このため、コンパイラの多くはアセンブリ言語を出力する。アセンブリ言語のコードは**アセンブラ** (assembler) によってさらに機械語に変換されて実行される。VM とアセンブラ (アセンブリ言語) については、後の例題で実際に扱ってみる。

2.2 抽象構文木の解釈実行

前回は「抽象構文木をさまざまな形で打ち出す」ことだけやったが、実は打ち出す代わりに「その構文木で表されている動作を行う」ことにすると、解釈実行系として動作させることができる。実際にやってみよう。なお、ツリーは次のような (簡単なサンプル言語の) プログラムに対応するものとしている:

```
read x; x = x + 1; print x;
```

ということは、この例題は前回の演習 6(a) の回答例を兼ねているのだった。ただし、前回は`'i'`(整数定数) のノードの `left` には `String` を入れていたが、今度は計算もするので `Integer` を入れるように変えてある。

なお、この言語では (さぼっていて) 変数名は英小文字 1 文字だけということにしている。そこで、26 要素の整数配列 `var` を用意し、ここに変数 a〜z の値をそれぞれ入れるということにしている (使っていない変数の分も構わず取っている)。

```
import java.io.*;

public class R8Sample1 {
  static BufferedReader in;
  static int[] var = new int[26];
  public static void main(String[] args) throws Exception {
    in = new BufferedReader(new InputStreamReader(System.in));
    Tree prog =
      new Tree(';',
        new Tree('r', new Tree('v',"x")),
        new Tree(';',
          new Tree('=', new Tree('v',"x"),
            new Tree('+', new Tree('v',"x"), new Tree('i', new Integer(1)))),
            new Tree(';',
              new Tree('p', new Tree('v',"x")))));
    prog.exec();
  }
  static class Tree {
    char op; Object left, right;
    public Tree(char o, Object l, Object r) { op = o; left =l; right = r; }
    public Tree(char o, Object l) { op = o; left =l; }
    public int exec() throws Exception {
      if(op == ';') {
        ((Tree)left).exec(); if(right != null) { ((Tree)right).exec(); }
      } else if(op == 'i') {
        Integer i = (Integer)left; return i.intValue();
      } else if(op == 'v') {
        String s = (String)left; return var[s.charAt(0)-'a'];
      } else if(op == 'r') {
        String s = (String)(((Tree)left).left); System.out.print("input> ");
        var[s.charAt(0)-'a'] = new Integer(in.readLine()).intValue();
      } else if(op == 'p') {
        System.out.println(((Tree)left).exec());
      } else if(op == '=') {
        String s = (String)(((Tree)left).left);
        return var[s.charAt(0)-'a'] = ((Tree)right).exec();
      } else if(op == '+') {
```

```
        return ((Tree)left).exec() + ((Tree)right).exec();
      }
      return 0;
    }
    public String toString() { } // 今回は使わないので打たなくてよい
      if(op == 'i' || op == 'v') { return ""+left; } //☆
      String l = ""; if(left != null) l = ((Tree)left).toString() + " ";
      String r = ""; if(right != null) r = " " + ((Tree)right).toString();
      return "(" + l + op + r + ")";
    }
  }
}
```

クラス Tree は追加したメソッド exec() 以外は同じ…だが、上記の整数を Integer オブジェクトに変更した都合で toString() の☆のところもキャストの代わりに文字列連結で文字列を作っている。ただし動作させるだけなら使わないので打たなくてよい (が、木構造をちゃんと作れているかどうかチェックしようと思ったら必要かも)。

さて、本題の追加した exec() を見てみよう (なぜ exec() に thorws Exception が必要なのかは次節で説明するのでとりあえず打っておいてください)。このメソッドはその木が表すプログラムの動作を実行し、式など実行結果があるものについてはその結果の値を返す (すべて整数型)。内容はノードの種別ごとに枝分かれするので、それぞれについて説明する。

- 「;」(連続実行) — 左の子を実行し、続いて右の子を実行。
- 「i」(整数定数) — intValue() で整数値を取り出して返す。
- 「v」(変数) — 配列 var のその変数のところの値を返す。
- 「r」(read 文) — まず値を読み込む。そして左の子は「v」なので、その変数に読み込んだ値を格納する。
- 「p」(print 文) — 左の子が打ち出すべき式なので、その値を計算して打ち出す。
- 「+」(足し算) — 左の子の値と右の子の値を計算し、その和を自分の値として返す。

意外と簡単だったでしょう?

演習 1 上のプログラムをそのまま打ち込んで動かせ (やや長いが、toString() は使わないので打たなくてよい)。動いたら、次のようなプログラムの木構造に取り替えて動かせ。

a. `read x; read y; z = x + y; print z;`

b. `read x; read y; read z; pirnt x + y + z;`

c. `read x; x = x + x; x = x + x; x = x + x; print x;`

演習 2 さらに次のようなプログラムについても同様に行え。ただしそのためには必要なノード種別の実行機能を追加する必要がある。なお、loop(N) ... は本体を N 回実行するループ文で、ノード種別'l' で表す、ということにしておく。

a. `read x; read y; print x * y;`

b. `read x; n = 0; r = 1; loop(x) { n = n + 1; r = r * n; } print r;`

c. `read x; read y; ifminus(x - y) { x = y; } print x;`

3 例外処理

3.1 例外と try 文

これまで、さまざまな「エラー」が起きるとプログラムは止まってしまい、その状況を表すメッセージが表示される、という状況でプログラムを開発してきた。しかし実は、「エラー」の一部は受け止めてプログラムの実行を続行してもよい。このようなプログラム言語の機能全般を**例外処理**と呼んでいる。これについて説明しよう。

そもそも、プログラムを記述していると「ここで入力に数字でないものが来たら」とか「ここでファイルが存在しなかったら」など、さまざまなエラーにつながる「もしかしたら」が存在する。もちろん、それらについて全部 if 文など

で枝分かれして対処すればよいのだが、プログラムが非常に長く繁雑になる。この講義では例題を短くするため、そのような処理を全部ほったらかしたままやってきたが、本来ほめられたことではない。

ところで、きちんとチェックするとしても、if 文を使って枝分かれする方法にはいくつかの弱点がある:

- 本来の処理の枝分かれとそのような例外的な場合を調べる枝分かれとの区別がつきにくい。
- 枝分かれによってプログラム本来の流れが分かりにくくなる。
- 手続きの中で「例外的な場合」が見つかったとき、それを呼び出し元に知らせるのが簡単でない。たとえば論理値で成功/失敗を返せばいいと思うかも知れないが、手続きは値を 1 つしか返せないので、成功/失敗に値を使ってしまうと他の用途に返値が使えなくて不便である。

これらの問題に対処するため、最近の言語ではこのような例外的なことがらをを専用の例外処理機構によって扱う。

具体的には、一連の処理の中で起きた例外的なことがら (以下「例外」) を 1 箇所に集めて、そこでまとめて処理できるようにしている。Java ではそのために、例外を受け止めるための構文である try 文というものを使う。具体的には、try 文は

```
try {
  文 ... (1)
} catch(例外クラス名 変数) {
  文 ... (2)
}
```

という形をしていて、最初の (1) の部分にある文のならびの中で発生した例外を受け止めることができる。具体的には、例外が起きると実行は (2) の文のならびに「ジャンプ」して、この部分の文を最後まで実行したら try 文全体の処理が終わって次の文に進む。例外が起きなければ (1) の中の文は全部実行されるが、(2) の文は実行されないことになる。つまり (2) の部分はエラー処理専用の動作ということになる。

3.2 例外クラス

ここで「例外クラス」はエラーの種類を表すもので複数あるが、次のように階層構造に分類されている。

```
Throwable --- すべての例外やエラーの全体
  Error --- 回復不可能なエラー
    ClassFromatError --- .class ファイルが変である
    NoClassDefFoundError --- .class ファイルが見つからない
    ... その他いろいろ ...
  Exception --- 例外的なできごと全般
    RuntimeException --- 実行時エラー全般
      NumberFormatException --- 文字列が数値の形式になってない
      NullPointerException --- null 値に対してメソッドを呼ぼうとした
      IndexOutOfBoundsException --- 配列の添字範囲外をアクセスした
      ... その他いろいろ ...
    IOException --- 入出力エラー
    IntrruptedException --- 一時停止中に割り込みが起きた
    IllegalAccessException --- クラスの内容を不正にアクセスしようとした
    ... その他いろいろ ...
```

通常は受け止めて処理するのは Exception 以下なので、「例外クラス名」として Exception を指定することが多い。ただし、ある特定の例外だけ別に扱いたければ次のように try 文を入れ子にして使うことができる。

```
try {
  ... ※ A
  try {
    ... ※ B
    ...
  } catch(NumberFormatException e) {
    (1) 数値の形式が間違っていた場合の処理
  }
  ... ※ C
} catch(Exception e) {
  (2) その他すべての例外の処理
}
```

ここで、※ B で NumberFormatException 例外が起きた場合には (1) の位置にある処理を実行するが、それ以外の例外は (2) の位置で処理する。※ A や※ C は内側の try 文の範囲外なので、すべての例外を (2) で処理する。[1]

3.3 例外を受け止めたところの処理

「例外を処理する」具体的な内容としては、次のようなものが考えられる。

- 何も動作を書かない — 単に例外が起きても無視して先の処理へ進みたい場合は何も動作を書かなくてもよい。
- 簡単なエラーメッセージを表示する。System.err.println(...) でメッセージを出力すればよい。なお、System.err.println() は System.out.println() とほぼ同様だが、エラー表示専用の出力ストリームになっている。
- 例外クラスすべてが共通に持っているメソッド `printStackTrace()` を呼び出して詳しいエラー状況を表示する。

実はこれまでエラーが自動的に表示される場合は 3 番目の `printStackTrace()` による表示が行われていた。
では、エラーを処理する簡単な例題を見てみよう。

```
import java.io.*;

public class R8Sample2 {
  public static void main(String[] args) {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    while(true) {
      try {
        System.out.print("x> ");
        String s = in.readLine();
        if(s.equals("")) break;
        int x = (new Integer(s)).intValue();
        System.out.print("y> ");
        int y = (new Integer(in.readLine())).intValue();
        System.out.println("x + y = " + (x+y));
      } catch(Exception e) {
        System.err.println("Oops! some error occured...");
        e.printStackTrace();
        System.err.println("Continue...");
      }
    }
  }
}
```

この場合、たとえば入力に数字でないものを入れると、NumberFormatException が発生するが、それをループの内側で catch で受け止めているのでプログラムはエラーで終了せず、次のデータを入れることができる。動いている様子を見てみよう。

```
% java R8Sample2
x> 3
y> 5
x + y = 8     ← OK
x> 3
y> a          ←まずいデータ
Oops! some error occured...
java.lang.NumberFormatException: a
        at java.lang.Integer.parseInt(Integer.java:426)
        at java.lang.Integer.<init>(Integer.java:567)
        at R8Sample2.main(R8Sample2.java:13)
Continue...
x> b          ←まずいデータ
Oops! some error occured...
java.lang.NumberFormatException: b
        at java.lang.Integer.parseInt(Integer.java:426)
        at java.lang.Integer.<init>(Integer.java:567)
        at R8Sample2.main(R8Sample2.java:11)
Continue...
x> 1
```

[1] だったら Exception を受け止めれば全部受け止められるのだからそれだけ書けばいいとか思いますか? 無節操に全部受け止めるのは「どのような例外が出るかちゃんと考えていない」兆候なのであまりよくないと言えます。ただ、他人に提供するプログラムで「予定外のことが起きた場合でも動き続けたい」なら Exception を受け止めてそれをエラー表示するのはよいことだと考えます。受け止めて無視する、というのは避けてください。

```
y> 2
x + y = 3    ← OK
x>           ← [ret] でおしまい
%
```

このように、例外を自前で処理することで何かエラーがあっても終らずに続行することができる。

3.4 例外を投げる

ここまでは例外を受け止める話ばかりだったが、例外は自分で投げることもできる。たとえば処理していて正しくない事態に陥ったと思った時、そこで何かじたばたするより例外を投げてどこか別の場所でまとめて処理した方がきれいにできるのが普通なので、そのような時は自分で例外を投げれば良い。例外を投げるには、throw 文を使う。

```
throw 式;
```

ただし「式」は例外クラス (Throwable 以下) のオブジェクトを返すものでなければならない。とりあえずは次のような形で RuntimeException オブジェクトを投げればよいだろう (自分でもっと適切なクラスを選んだり新規に作ってもよいが)。

```
throw new RuntimeException("ほにゃららがまずい。");
```

また、先の catch 節での処理に次のものも追加しておこう。

- 一旦受け止めてエラーメッセージを出すが、さらに外側で処理してもらうため同じ例外を再度投げ直す。

この場合は次のような書き方になるだろう。

```
try {
  ...
} catch(Exception e) {
  System.err.println("....");
  throw e; // 投げなおす
}
```

3.5 throws 宣言

自分で投げた例外はこれまで通り catch で受け止めればよいが、自分のメソッド内では受け止めず、外側 (そのメソッドを呼び出した側) で受け止める場合にはメソッド定義の冒頭部分に

```
... メソッド名 (パラメタ…) throws 例外クラス名,... {
```

の形で宣言しておかなければならない。つまり「私はこれこれの例外を投げますよ」と予め明らかにしておかないと、そのメソッドを呼ぶ側で「心の準備」ができないので呼んでみたら知らない例外が投げられてびっくり、といったことになるからである。

実は、throws 節による宣言は、そのメソッドの中で throw で例外を投げる可能性がある場合だけでなく、そのメソッドの中で例外が発生するような操作を行って、なおかつその例外を catch しない場合にも必要である。というのは、catch しなかった例外はそのままメソッド呼び出し側に伝わって行くことになるから。

これでようやく、第 1 回のおまじないの謎が解けることになる。つまり、

```
public static void main(String[] args) throws Exception { ...
```

というのは、この中で例外 (実は in.readLine() を呼ぶと IOException が投げられることがある) が発生してそれを捕まえないので外側にもそれが伝わりますよ、ということを断っているのだった。

ただし、この規則にも「例外」があって、Error と RuntimeException およびそれの子孫の例外については throws 節で断らなくてもよい。というのは、これらの例外はあらゆる場所で発生し得るため、いちいち断っていると大変すぎるからである。上の「おすすめ」で RuntimeExcpetion を投げればいいと書いたのは、この例外であればとくに断らないでも投げることができるので簡単に済むからだった (手抜きでよくないとも言えるけど)。

…大変長い説明ですいませんでしたが、一応きちんと説明するとこういうことになってしまうのでした。

4　抽象構文木から実行へ (2)

4.1　i386 アセンブリ言語の出力

前半の説明で抽象構文木を解釈実行するインタプリタを示したが、やっぱり CPU が実行する命令を出力するコンパイラの方がかっこよさそうですよね? そこで、Windows マシンに使われている i386 系 CPU(IA32) 用のアセンブリ言語を出力してみよう。もちろん、特定 CPU 用の命令を出すのだから、別の種類の CPU では動きません。そして ECC 環境では CPU が PowerPC なので動きません。PPC 命令を出すのはちょっと面倒なので、学校のマシンしか使えない人や自宅 Mac な人はとりあえず見物だけということで。

このアセンブリ言語をアセンブルして実行可能にするのには GCC と呼ばれる開発環境を使う。Unix/Linux 環境には GCC は最初から備わっているが、Windows の場合は Cygwin と呼ばれる Unix 互換環境を入れれば動かすことができる (入れるときに何を入れるか選べるので、GCC ををを忘れずに入れてください)。そんなものを入れたりできないという人もやはり見物だけということで。

見物だけでは実感が湧かないかも知れませんが、とりあえず気分だけ味わって、次節以降の簡単な VM とアセンブラの方をやってください。

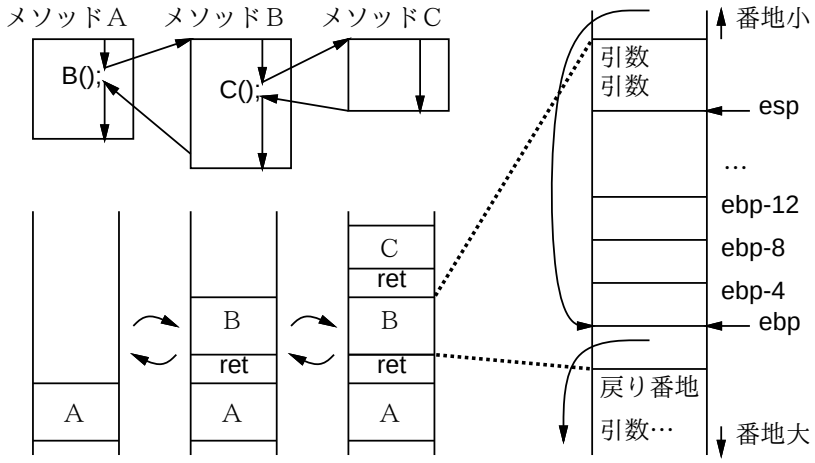

図 4: i386 での関数の使用領域

i386 を含む現在の多くの CPU のコードでは、ある関数が使う局所変数などのメモリ領域は図 4 左のようにスタックの一部の形で取られていて、関数が呼ばれるとその領域が割り当てられ (push され)、return すると解放される (pop される)。これは、関数が「最後に呼び出されたものが最初に return する」性質を持つので、その領域はスタックで管理するのがぴったりだからである (また、こうしておくと再帰手続きが自然に実現できる)。

1 つの関数呼び出しが使う部分を詳しく見ると、図 4 右のような構造をしている。その詳細は次の通り:

- レジスタ ebp が 1 つ前の関数実行中の ebp の格納位置を指している。
- ebp-4、ebp-8、ebp-12、…にこの関数が使う変数を入れる。アセンブラではこれらのメモリ位置は「`-4(%esp)`」のような書き方で指定する。
- レジスタ esp が変数領域の終わりを指している (関数入口でそうなるように esp を引き算して設定する)。
- 他の関数を呼ぶ時の引数は esp を指定した push 命令で積む。その後 call 命令を実行することで戻り番地 (関数呼び出しの次の命令の番地) がスタックに積まれ、関数のコードにジャンプすることで関数を呼び出せる。
- 命令列 leave、ret で ebp と esp を関数呼び出し前の値に復元して戻り番地へ戻れる。

関数の入口/出口のためのの命令列は「esp を引き算する量 (変数の数)」以外は固定なので決め打ちで出力する。関数本体で使う命令は次の通り (除算は説明が長くなるので略)。

- `pushl` 値 — 値を esp の位置に格納し、esp を 4 減らす。つまりスタックに積む動作。「値」はレジスタ (`%eax` 等) でも定数 (`$10` 等) でもメモリ上の位置 (`-8(%ebp)` 等) でもよい (以下同様)。
- `call` 名前 — 指定した名前の関数 (サブルーチン) を呼ぶ。

```
 .section .rodata       ←以下定数
.L0: .string "input> " ←文字列その 1
.L1: .string "%d"      ←文字列その 2
.L2: .string "%d\n"    ←文字列その 3
 .text                 ←以下コード
 .p2align 2,,3         ←境界合わせ指令
.globl main            ← main が
 .type main,@function  ←関数だとアセンブラに指示
main:                  ←ここが main の開始
 pushl %ebp            ←ここから関数入口処理
 movl  %esp,%ebp
 subl  $8,%esp
 andl  $-16,%esp
 subl  $28,%esp        ←変数のぶんだけ esp をずらす
 pushl $.L0            ←文字列その 1 の番地を積んで
 call  printf          ← printf を呼ぶ
 addl  $4,%esp         ← esp を積んだ分だけ戻す
 lea   -4(%ebp),%eax   ←変数 x の番地を
 pushl %eax            ←積む
 pushl $.L1            ←文字列その 2 の番地を積む
 call  scanf           ← scanf を呼ぶ
 addl  $8,%esp         ← esp を積んだ分だけ戻す
 movl  $0,%eax         ← eax を 0 に
 movl  %eax,-8(%ebp)   ←変数 n に eax の値を入れる
 movl  $1,%eax         ← eax を 1 に
 movl  %eax,-12(%ebp)  ←変数 r に eax の値を入れる
 movl  -4(%ebp),%eax   ← x を eax に
 movl  %eax,-24(%ebp)  ← eax を t24 に
.L3:
 decl  -24(%ebp)       ← t24 を 1 減らす
 jl  .L4               ←マイナスなら L4 へ飛ぶ
 movl  $1,%eax         ← 1 を eax に
 movl  %eax,-16(%ebp)  ← eax を t16 に
 movl  -8(%ebp),%eax   ← n を eax に
 addl  -16(%ebp),%eax  ← t16 の値を足す
 movl  %eax,-8(%ebp)   ← eax を n に格納
 movl  -8(%ebp),%eax   ← n を eax に
 movl  %eax,-20(%ebp)  ← eax を t20 に
 movl  -12(%ebp),%eax  ← r を eax に
 imull -20(%ebp),%eax  ← t20 を掛ける
 movl  %eax,-12(%ebp)  ← eax を r に格納
 jmp .L3               ← L3 に飛ぶ
.L4:                   ← L4 はここ
 movl  -12(%ebp),%eax  ← r を eax に
 pushl %eax            ← eax を積む
 pushl $.L2            ←文字列その 3 の番地を積む
 call  printf          ← printf を呼ぶ
 addl  $8,%esp         ← esp を積んだ分だけ戻す
 leave                 ←以下関数出口処理
 ret
```

図 5: i386 アセンブリ言語による出力

- j 番地 — 指定番地にジャンプ
- movl 値, 場所 — 値を場所に格納す。代入文のようなもの。
- lea 場所,%eax — 「場所」はメモリ上の位置でなければならず、そのメモリ番地をレジスタ eax に入れる。
- addl 値, 場所 — 場所に入っている値に指定した値を加算。subl だと減算、imull だと乗算になる。
- incl 場所 — 場所に入っている値を 1 増やす。decl だと 1 減らす。

add や inc などの演算命令では、演算した結果の正/負/零が条件コードレジスタに記録される。j 命令のかわりに jl、jle、jg、jge、je、jne を使うと、この条件コードを参照して結果が負 (less)、0 以下 (less or equal)、正 (greater)、0 以上 (greater or equal)、零 (equal)、非零 (not equal) の時だけジャンプが起きる。呼び出すサブルーチンは次の 3 種類だけ (C 言語用のルーチン)。

```
printf("input> ");       --- 文字列出力
scanf("%d", 変数の番地) --- 整数入力
printf("%d\n", 値)       --- 整数出力
```

この 3 つの呼び出しに使う文字列はアセンブラの機能を使って予め格納しておく。なお、引数 2 個の場合、後ろのものを先に積む規約になっている。サンプルとして使う「簡単な言語のプログラム」は今度は次のもの (階乗の計算)。

```
read x; n = 0; r = 1; loop(x) { n = n + 1; r = r * n; } print r;
```

これを例題プログラムで i386 コードに変換したものを図 5 に示す。t24 とか t16 とかは作業用に自動で割り当てた変数を意味する。

こうして読むとえらく無駄の多いコードだが、素朴なコード生成だとこのようなもので普通である。ではこれを生成するプログラムを示す。26 個の変数の割り付け位置を記録するのに配列 var を使い、また次に使用する変数の位置を vpos で覚え、次に出すラベルの番号を labno で覚える。これらは main() と後の部分で共通に使うため外側に書いてある。

main() では、まず木構造を組み立て、その後 prepare() を呼んで変数の場所を割り付ける。それから決め打ち部分を出力するが、このとき vpos を用いてスタックを減らす量を出力する。それからコード本体を出力し、最後の決め打ち部分を出力する。

```
public class R8Sample3 {
  static int[] var = new int[26];
  static int vpos = 4, labno = 3;
  public static void main(String[] args) throws Exception {
    Tree prog =
```

```
     new Tree(';',
      new Tree('r', new Tree('v',"x")),
      new Tree(';',
       new Tree('=', new Tree('v',"n"), new Tree('i', new Integer(0))),
       new Tree(';',
        new Tree('=', new Tree('v',"r"), new Tree('i', new Integer(1))),
        new Tree(';',
         new Tree('l', new Tree('v', "x"),
          new Tree(';',
           new Tree('=', new Tree('v',"n"),
            new Tree('+', new Tree('v',"n"), new Tree('i', new Integer(1)))),
           new Tree(';',
            new Tree('=', new Tree('v',"r"),
             new Tree('*', new Tree('v',"r"), new Tree('v',"n")))))),
           new Tree(';',
            new Tree('p', new Tree('v', "r")))))));
    prog.prepare();
    System.out.println(" .section .rodata");
    System.out.println(".L0: .string \"input> \"");
    System.out.println(".L1: .string \"%d\"");
    System.out.println(".L2: .string \"%d\\n\"");
    System.out.println(" .text");
    System.out.println(" .p2align 2,,3");
    System.out.println(".globl main");
    System.out.println(" .type main,@function");
    System.out.println("main:");
    System.out.println(" pushl %ebp");
    System.out.println(" movl  %esp,%ebp");
    System.out.println(" subl  $8,%esp");
    System.out.println(" andl  $-16,%esp");
    System.out.println(" subl  $" + vpos + ",%esp");
    prog.emit();
    System.out.println(" leave");
    System.out.println(" ret");
  }
```

クラス Tree の冒頭部分はこれまでと同じだが、ノードに作業用変数を割り当てることがあるので、その場合にその位置を記録する変数 (フィールド)offset を追加している。関数 dovar() は指定した変数 (1 文字) がまだ割り付けてなければその割り付け位置を配列 var に記録し、vpos を次の位置まで増やす。

```
  static class Tree {
    char op; Object left, right; int offset;
    public Tree(char o, Object l, Object r) { op = o; left =l; right = r; }
    public Tree(char o, Object l) { op = o; left =l; }
    private static void dovar(String s) {
      if(var[s.charAt(0)-'a']==0) { var[s.charAt(0)-'a'] = vpos; vpos += 4; }
    }
```

メソッド prepare() は再帰呼び出しで木をたどるが、その中で変数が現れるところについては dovar() を呼び出して変数の位置を割り付ける。また、「+」「*」のノードについては、作業用変数を割り当てる。なぜ作業用変数が必要かというと、たとえば「+」の左も右も込み入った式の場合、まず左を計算して、その結果はどこかに覚えて置かないと右を計算している間に失われてしまう (レジスタはすぐ次の計算に使う) から。あと、ループ文でも作業変数を使う。

```
    public void prepare() {
     if(op == ';') {
        ((Tree)left).prepare(); if(right != null) { ((Tree)right).prepare(); }
      } else if(op == 'v') {
        String s = (String)left;  dovar(s);
      } else if(op == 'r') {
        String s = (String)(((Tree)left).left); dovar(s);
      } else if(op == 'p') {
        ((Tree)left).prepare();
      } else if(op == '=') {
        String s = (String)(((Tree)left).left); dovar(s);
        ((Tree)right).prepare();
      } else if(op == '+' || op == '*' || op == 'l') {
        ((Tree)left).prepare(); ((Tree)right).prepare();
        offset = vpos; vpos += 4;
      }
    }
```

ではいよいよコード出力するメソッド emit() を見てみる。

```
    public void emit() {
     if(op == ';') {
        ((Tree)left).emit(); if(right != null) { ((Tree)right).emit(); }
      } else if(op == 'i') {
        Integer i = (Integer)left;
        System.out.println(" movl  $" + i.intValue() + ",%eax");
      } else if(op == 'v') {
        String s = (String)left;
        System.out.println(" movl  -" + var[s.charAt(0)-'a'] + "(%ebp),%eax");
      } else if(op == 'r') {
        String s = (String)(((Tree)left).left);
        System.out.println(" pushl $.L0");
        System.out.println(" call  printf");
        System.out.println(" addl  $4,%esp");
        System.out.println(" lea   -" + var[s.charAt(0)-'a'] + "(%ebp),%eax");
        System.out.println(" pushl %eax");
        System.out.println(" pushl $.L1");
        System.out.println(" call  scanf");
        System.out.println(" addl  $8,%esp");
      } else if(op == 'p') {
        ((Tree)left).emit();
        System.out.println(" pushl %eax");
        System.out.println(" pushl $.L2");
        System.out.println(" call  printf");
        System.out.println(" addl  $8,%esp");
      } else if(op == '=') {
        String s = (String)(((Tree)left).left);
        ((Tree)right).emit();
        System.out.println(" movl  %eax,-" + var[s.charAt(0)-'a'] + "(%ebp)");
      } else if(op == '+') {
        Tree t1 = (Tree)left, t2 = (Tree)right;
        t2.emit();
        System.out.println(" movl  %eax,-" + offset + "(%ebp)");
        t1.emit();
        System.out.println(" addl  -" + offset + "(%ebp),%eax");
      } else if(op == '*') {
        Tree t1 = (Tree)left, t2 = (Tree)right;
        t2.emit();
        System.out.println(" movl  %eax,-" + offset + "(%ebp)");
        t1.emit();
        System.out.println(" imull -" + offset + "(%ebp),%eax");
      } else if(op == 'l') {
        Tree t1 = (Tree)left, t2 = (Tree)right;
        int l = labno; labno += 2;
        t1.emit();
        System.out.println(" movl  %eax,-" + offset + "(%ebp)");
        System.out.println(".L" + l +":");
        System.out.println(" decl  -" + offset + "(%ebp)");
        System.out.println(" jl  .L" + (l+1));
        t2.emit();
        System.out.println(" jmp .L" + l);
        System.out.println(".L" + (l+1) +":");
      }
    }
    public String toString() {
      if(op == 'i' || op == 'v') { return ""+left; }
      String l = ""; if(left != null) l = ((Tree)left).toString() + " ";
      String r = ""; if(right != null) r = " " + ((Tree)right).toString();
      return "(" + l + op + r + ")";
    }
  }
}
```

- 'i' — その定数を movl で eax に持って来る。
- 'v' — その変数の値を movl で eax に持って来る (read 文や代入はこれとは別扱いでコードを出す)。

- 'r' — printf と scanf で値を読み込む。left に入っている変数の位置を scanf に渡すことで、そこの場所に値が入る。
- 'p' — 式の計算コードを生成すると、結果は eax に値が入るので、それを積んで printf で表示。
- '+'/'-' — まず右側の式を計算して、その結果を作業変数に格納する。続いて左側の式を計算して、それに作業変数に覚えておいた値を足す/掛ける。
- 'l' — ラベルが 2 つ必要なので、現在の labno を別の変数におぼえてから labno を 2 つ増やす。まずかっこ内の値を計算し、それを作業変数に入れる。次に前のラベルを出力。次に作業変数を decl で 1 減らし、マイナスなら後のラベルに飛ぶ。あとはループ本体を出力し、前のラベルへのジャンプを出力し、後のラベルを出力する。

これを動かすといきなりアセンブリ言語コードが出力されるので、それをファイルに取り、gcc コマンドで機械語に変換すると実行可能になる。

```
% javac R8Sample3.java
% java R8Samle3 >test.s  ←アセンブリ言語コードは「.s」
% gcc test.s             ←アセンブラで処理
% a.exe        ← Windows+cygwin の場合。Unix だと「a.out」
input> 5       ←実行開始し入力要求
120            ←確かに階乗の計算ができる
%
```

なかなか大変だけど、いかにもコンパイラという感じで楽しいでしょう?

演習 3 もっと別のプログラムのコードを生成させてみよ。

演習 4 ループ文以外の制御構造を実現してみよ。

演習 5 無駄な命令を削減して実行効率を向上させてみよ。

なお、これらの演習の「成果確認」は生成されたコードを見てチェックするだけでもよいです (i386 の GCC を入れるのは大変でしょうから)。または、次節で説明する仮想マシン用アセンブリ言語コードを対象にしてもよいです (それなら仮想マシンまで打ち込めば動かせます)。

4.2 仮想マシン用アセンブリ言語コードの出力

前節 i386 のアセンブリ言語だと動かして見られない人もいるし、実際の機械語がどうなっているかという感じが持てないので、もう 1 つ簡単な仮想マシンを想定し、そのアセンブリ言語を生成することにした。この仮想マシンはすべての命令が 32 ビットで、上 16 ビットが命令コード、下 16 ビットがオペランド (演算対象) である。メモリも 1 語 32 ビット単位で 0 番地から番地がついている。レジスタは**アキュムレータ** (ac) と呼ばれるレジスタが 1 個だけあり、これですべての演算を行う。このため i386 のように命令で「どのレジスタ」を指定する必要がない。命令としては次のものがある (番号は命令コードを意味し、また d でオペランドの指定値を表す)。命令は 0 番地から 1 命令ずつ順に実行するが、分岐命令が実行された場合は次の実行命令は分岐先の命令になる。

- load(1) — d 番地の内容を ac レジスタに転送。
- loadi(2) — 数値 d を ac レジスタに転送 (このようにオペランド値が番地でなく演算に使う値である場合に**即値** (immidiate value) と呼ぶ。
- store(3) — ac レジスタの内容を d 番地に格納。
- call(4) — まとまった機能を呼び出すための特別な命令。d が 0 のときは整数を入力し ac に入れる。d が 1 のときは ac の値を画面に出力。
- stop(5) — d を表示してプログラムの実行を終了する。
- add(11)、sub(12)、mul(13)、div — ac の内容に d 番地の内容を加算/減算/乗算/除算。
- addi(11)、subi(12)、muli(13)、divi(14) — ac の内容に値 d を加算/減算/乗算/除算。
- jmp(30)、jl(31)、jle(32)、jg(33)、jge(34)、je(35)、jne(36) — d 番地への分岐命令。jmp は常に分岐するがそれ以外は最後に実行した演算命令の結果が負/零以下/正/零以上/零/非零の時だけ分岐。

```
call 0          L1:             add 104         L2:
store 101        load 106       store 102        load 103
loadi 0          subi 1         load 102         call 1
store 102        store 106      store 105        stop 0
loadi 1          jl L2          load 103
store 103        loadi 1        mul 105
load 101         store 104      store 103
store 106        load 102       jmp L1
```

図 6: 簡単な仮想マシンのアセンブリ言語による出力

このアセンブリ言語を出力するコードは、emit() 以外は「vpos の初期値は 101」「labno の初期値は 1」「vpos は 1 ずつ増やす (1 番地が 1 語なので)」の 3 点を除き先の例と一緒なので emit() だけを示す。

```
public void emit() {
 if(op == ';') {
    ((Tree)left).emit(); if(right != null) { ((Tree)right).emit(); }
  } else if(op == 'i') {
    Integer i = (Integer)left;
    System.out.println(" loadi " + i.intValue());
  } else if(op == 'v') {
    String s = (String)left;
    System.out.println(" load " + var[s.charAt(0)-'a']);
  } else if(op == 'r') {
    String s = (String)(((Tree)left).left);
    System.out.println(" call 0");
    System.out.println(" store " + var[s.charAt(0)-'a']);
  } else if(op == 'p') {
    ((Tree)left).emit();
    System.out.println(" call 1");
  } else if(op == '=') {
    String s = (String)(((Tree)left).left);
    ((Tree)right).emit();
    System.out.println(" store " + var[s.charAt(0)-'a']);
  } else if(op == '+') {
    Tree t1 = (Tree)left, t2 = (Tree)right;
    t2.emit();
    System.out.println(" store " + offset);
    t1.emit();
    System.out.println(" add " + offset);
  } else if(op == '*') {
    Tree t1 = (Tree)left, t2 = (Tree)right;
    t2.emit();
    System.out.println(" store " + offset);
    t1.emit();
    System.out.println(" mul " + offset);
  } else if(op == 'l') {
    Tree t1 = (Tree)left, t2 = (Tree)right;
    int l = labno; labno += 2;
    t1.emit();
    System.out.println(" store " + offset);
    System.out.println("L" + l +":");
    System.out.println(" load " + offset);
    System.out.println(" subi 1");
    System.out.println(" store " + offset);
    System.out.println(" jl L" + (l+1));
    t2.emit();
    System.out.println(" jmp L" + l);
    System.out.println("L" + (l+1) +":");
  }
}
```

このプログラム動かすといきなりアセンブリ言語コードが出力されるので、たとえば

```
java R8Sample4 >test.asm
```

のようにして結果をファイルに保存する。先と同じ木 (階乗の計算) のアセンブリ言語を生成したものを図 6 に示す。命令は変わっても基本的な構造は同じであることが分かると思う。

4.3 仮想マシン用アセンブラ

では、前節のプログラムが生成したアセンブリ言語コードを機械語 (ビット列のコード) に直すプログラム、つまりアセンブラを作ってみよう。といっても原理は簡単で、先頭から 1 命令ずつ、命令語とオペランドを取り出して 1 語に組み立てて行くだけ。組み立てた語はすぐ出力するのではなく、一旦配列に入れておき、最後にまとめて出力する。

ただし、ラベルがあった場合はその番地を覚えておき、ラベル参照があったらその番地の値で置き換える。ここで問題なのは、ジャンプ命令が先にあってラベルが後にあると、そのラベルの場所まで処理しないとラベルが何番地なのか分からないこと (**前方参照**の問題)。これを解決するために、命令列を 2 回読んで (**2 パス**方式)、まったく同じに処理する。すると 2 回目はラベルの番地は分かっているから OK になる。

命令の名前と命令コード、ラベルの名前と番地は表で管理する必要がある。このため、文字列をキー、整数を値とする、線形探索を用いた表のクラス `StrIntTable` を用意して用いる (検索で見つからないときは- 1 を返すようにしている)。

では先頭から見てみよう。まず冒頭で命令表、ラベル表を用意し、命令表にはすべての命令のコードを登録する。そのあと、コードを入れるための場所 (`Memory` クラスとした) を用意し、2 回アセンブリ命令を読ませる (1 回読ませた後で入れる場所を一旦 0 に戻して再度読ませる)。読ませ終わったらコードを出力する。

```
import java.io.*;

public class R8Sample5 {
  public static void main(String[] args) throws Exception {
    StrIntTable optbl = new StrIntTable(100), label = new StrIntTable(100);
    optbl.put("load",  1); optbl.put("loadi", 2); optbl.put("store", 3);
    optbl.put("call",  4); optbl.put("stop",  5);
    optbl.put("add",  11); optbl.put("sub",  12); optbl.put("mul",  13);
    optbl.put("div",  14);
    optbl.put("addi", 21); optbl.put("subi", 22); optbl.put("muli", 23);
    optbl.put("divi", 24);
    optbl.put("jmp",  30);
    optbl.put("jl",   31); optbl.put("jle",  32); optbl.put("jg",   33);
    optbl.put("jge",  34); optbl.put("je",   35); optbl.put("jne",  36);
    Memory vm = new Memory(1000);
    load(vm, args[0], optbl, label, 1); vm.setLoadPoint(0);
    load(vm, args[0], optbl, label, 2); vm.dump();
  }
```

メソッド `load()` がアセンブラの本体である。指定されたファイルを開いて 1 行ずつ読み、先頭が空白でなければ現在位置をラベルとして登録し、そうでなければ命令とオペランド部を取り出し (簡単のため区切りの空白はずべて 1 個だけということにした)、命令は命令コードに変換、オペランドは数字で始まるならそのまま数値に変換、そうでない場合はラベルとして検索してラベルの番地に変換し、両方を組み合わせて命令語を作ってメモリに格納する (「`<<`はシフト演算、`|`はビット単位の or 演算)。命令の未定義や 2 パス目でのラベルの未定義は例外処理を使って 1 箇所でメッセージを出力している。

```
  static void load(Memory vm, String fname,
                   StrIntTable optbl, StrIntTable label, int pass) {
    try {
      String line;
      BufferedReader asm = new BufferedReader(new FileReader(fname));
      while((line = asm.readLine()) != null) {
        try {
          if(line.charAt(0) != ' ') {  // label
            String l = line.substring(0, line.length()-1);
            label.put(l, vm.getLoadPoint());
          } else {
            line = line.substring(1); int i = line.indexOf(' ');
            String s1 = line.substring(0, i), s2 = line.substring(i+1);
            int op = optbl.get(s1), opd = 0;
            if(op < 0) { throw new Exception("undefined op: "+line); }
            if(Character.isDigit(s2.charAt(0))) {
              opd = new Integer(s2).intValue();
            } else {
              opd = label.get(s2);
              if(pass==2 && opd<0) {throw new Exception("undef label: "+line);}
            }
            vm.load(op << 16 | opd);
          }
        } catch(Exception ex) { System.err.println("?"+ex); }
      }
```

```
40000   1006a   30066   40001
30065   160001  10066   50000
20000   3006a   30069
30066   1f0017  10067
20001   20001   d0069
30067   30068   30067
10065   10066   1e0008
3006a   b0068   10067
```

図 7: 簡単な仮想マシンの機械語

```
    } catch(Exception ex) { System.err.println(ex); return; }
  }
```

Memory クラスは単に配列を持っていて 1 語ずつ命令を格納するだけ。メソッド dump では Integer.toString() の 2 引数版を使って 16 進数形式で内容を出力する。

```
  static class Memory {
    int[] mem; int ptr = 0;
    public Memory(int s) { mem = new int[s]; }
    public void load(int i) { mem[ptr] = i; ++ptr; }
    public void setLoadPoint(int i) { ptr = i; }
    public int getLoadPoint() { return ptr; }
    public void dump() {
      for(int i = 0; i < ptr; ++i) {
        System.out.println(Integer.toString(mem[i], 16));
      }
    }
  }
```

StrIntTable クラスはごく簡単な線形探索の表で、レコードを使う代わりに文字列の配列と整数の配列をペアで使うようにしてみた。

```
  static class StrIntTable {
    String[] keys; int[] vals; int count = 0;
    public StrIntTable(int s) { keys = new String[s]; vals = new int[s]; }
    public void put(String k, int v) {
      for(int i = 0; i < count; ++i) {
        if(keys[i].equals(k)) { vals[i] = v; return; }
      }
      keys[count] = k; vals[count] = v; ++count;
    }
    public int get(String k) {
      for(int i = 0; i < count; ++i) {
        if(keys[i].equals(k)) { return vals[i]; }
      }
      return -1;
    }
  }
}
```

これを使って先のアセンブリ言語コードを簡単な仮想マシンの機械語に変換できる。それには、アセンブリ言語コードを格納したファイル名を指定し、出力される機械語は別のファイルに保存する。

```
java R8Sample5 test.asm >test.bin
```

先のアセンブリ言語コードを変換したものを図 7 に示す。16 進数で表現されていると、結構読めるものだと思いますがどうですか?

4.4 仮想マシン

せっかく機械語まで作ったので、それを実行するための仮想マシンも当然作ってみよう。とりあえず、メモリは 1000 番地まであることにして仮想マシンを作り、ファイル名を指定してメモリ内に機械語を読み込み、0 番地から実行を開始させる。

```
import java.io.*;
```

```
public class R8Sample6 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    VirtualMachine vm = new VirtualMachine(1000);
    vm.load(args[0]); vm.run(in);
  }
```

メモリにロードするときには 16 進数の読み込みが必要になるが、これは `Integer.valueOf()` を使えばできる (API ドキュメントでチェックしておいてください)。メソッド `ck()` は演算後に結果の正/負/零を記録しておくためのもの。

```
  static class VirtualMachine {
    int[] mem;
    int ac, ptr;
    boolean zero, minus, plus;
    public VirtualMachine(int s) { mem = new int[s]; }
    public void load(String fname) throws Exception {
      String line; ptr = 0;
      BufferedReader in = new BufferedReader(new FileReader(fname));
      while((line = in.readLine()) != null) {
        mem[ptr] = Integer.valueOf(line, 16).intValue(); ++ptr;
      }
      in.close();
    }
    private void ck() {
      zero = (ac == 0); plus = (ac > 0); minus = (ac < 0);
    }
```

`run()` がプログラム実行の本体であり、`ptr` の番地から命令語を取り出し、命令コードとオペランドに分解し、`ptr` は次の命令の番地に進めておく。あとは命令の種類ごとにすべて分岐して処理をする (call だけ長いので枝分かれの最後に置いた)。

```
    public void run(BufferedReader in) throws Exception {
      ptr = 0; ac = 0;
      while(true) {
        int inst = mem[ptr], op = inst>>16, opd = inst&0xFFFF;
        if(op == 1)      { ac = mem[opd]; } // load
        else if(op == 2) { ac = opd; }      // loadi
        else if(op == 3) { mem[opd] = ac; } // store
        else if(op == 5) { return; }                // stop
        else if(op == 11) { ac += mem[opd]; ck(); } // add
        else if(op == 12) { ac -= mem[opd]; ck(); } // sub
        else if(op == 13) { ac *= mem[opd]; ck(); } // mul
        else if(op == 14) { ac /= mem[opd]; ck(); } // div
        else if(op == 21) { ac += opd; ck(); }      // addi
        else if(op == 22) { ac -= opd; ck(); }      // subi
        else if(op == 23) { ac *= opd; ck(); }      // muli
        else if(op == 24) { ac /= opd; ck(); }      // divi
        else if(op == 30) { ptr = opd; }            // jmp
        else if(op == 31) { if(minus) { ptr = opd; } }  // jl
        else if(op == 32) { if(!plus) { ptr = opd; } }  // jle
        else if(op == 33) { if(plus) { ptr = opd; } }   // jg
        else if(op == 34) { if(!minus) { ptr = opd; } } // jge
        else if(op == 35) { if(zero) { ptr = opd; } }   // je
        else if(op == 36) { if(!zero) { ptr = opd; } }  // jne
        else if(op == 4) {                  // call
          if(opd == 0) {
            System.out.print("input> ");
            ac = new Integer(in.readLine()).intValue();
          } else if(opd == 1) {
            System.out.println(ac);
          }
        }
      }
    }
  }
}
```

これを使って先の機械語コードを実行させるには、ファイル名を指定して仮想マシンを起動すればよい。

```
  % javac R8Sample6.java
```

```
% java R8Sample6 test.bin
input> 5
120
STOP 0
%
```

演習 6 仮想マシン用アセンブラ出力プログラム、仮想マシン用アセンブラ、仮想マシンの各プログラムを打ち込んで、自作プログラムを実行させてみよ。動いたら Java で普通に書いたものと速度比較を試みよ。

演習 7 仮想マシンのように整数値 (や文字値) で多方向の枝分かれをする場合は、if-else if よりも **switch 文**を使う方が読みやすく効率がよい。これは次のような形をとる:

```
    switch(式) {
case 値: case 値: case 値:
      処理…
      break;
case 値: case 値:
      処理…
      break;
default:
      処理…
      break;
    }
```

冒頭の「式」の値を case ラベルに指定された値と照合し、あてはまる case ラベルのついている処理を実行する。break 文は switch から抜け出すのに使う (これが無いとそのまま次の処理に進んでしまう)。これを使って仮想マシンを書き直し、処理速度が向上したかどうか調べよ。

演習 8 プログラミング言語処理系に関係のある好きなプログラムを作ってみよ。

A　本日の課題 8A

「演習 1」または「演習 2」で動かしたプログラムを含む小レポートを今日中に久野までメールで送ってください。

1. Subject: は「Report 8A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 「演習 1」または「演習 2」で動かしたプログラムどれか 1 つのソース。
4. 以下のアンケートの回答。

   Q1. 抽象構文木の解釈実行について学びましたが、納得しましたか。

   Q2. アセンブラとか機械語についてどう思いましたか。

   Q3. 本日の全体的な感想と今後の要望をお書きください。

B　次々回までの課題 8B

次々回までの課題は「演習 2」〜「演習 8」の (小) 課題から 2 つ以上選んで報告することです。各課題のために作成したプログラムは複数ある場合もすべてレポートに掲載してください。レポートは 1/12(金) 授業開始時刻の **10 分前**までに久野までメールで送付してください。

1. Subject: は「Report 8B」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 1 つ目のプログラムのソース。
4. その説明と分析/考察。
5. 2 つ目のプログラムのソース。
6. その説明と分析/考察。
7. 下記のアンケートの回答。

   Q1. 簡単な言語処理系 (の実行部分) が作れるようになりましたか。

   Q2. 言語処理系とはどういうものか納得しましたか。

   Q3. 課題に対する感想と今後の要望をお書きください。