システムソフトウェア特論'19 # 3 言語処理系の構成

久野 靖 (電気通信大学)

2019.1.9

言語処理系の枠組み

言語処理系の分類

プログラミング言語処理系 (**programming language processor**、以下では語処理系と記す) とは、プログラミング言語の記述ないしソースコード (**source code**) を入力とし、そこに記述された動作 (==プログラムの動作) を実現するようなソフトウェア全般を指します。

大まかな分類として、言語処理系はインタプリタ (**interperter**) ないし解釈系とトランスレータ (**translater**) ないし翻訳系に 2 分できます。前者はソースコードを読み込んだ後、その動作を直接そのソフトウェアが実行するのに対し、後者はソースコードを同等の動作を行う目的コード (**object code**) ないしターゲットコード (**target code**) に変換して出力します。

ここで、目的コードの形式が機械語 (**machine language**) ないしそれに近い水準のものである場合に、その翻訳系のことをコンパイラ (**compiler**) と呼ぶのが慣わしです。コンパイラ以外の翻訳系としては、簡便な形で他の高水準言語に変換するプリプロセサ (**preprocessor**) などがあります。目的コードが他の高水準言語であっても、その言語のプログラムとして読めないような変換を行う場合はコンパイラに分類することもあります。

言語処理系の分類 (2)

コンパイラが特定CPUの命令を生成する場合、さまざまなマシンで動かすにはそれぞれのCPUの命令を出力するようにコンパイラを複数用意する必要がありますが、これはなかなか大変です。

コンパイラの中には、目的コードが実在のCPUの機械語ではなく、仮想的な(汎用性のある)機械語であるものもあります。この場合、その仮想的な機械語を実行するソフトウェアは仮想マシン(**virtual machine**)と呼ばれます。この方式であれば、コンパイラは1種類でよく、さまざまなCPU用に仮想マシンだけ用意すれば済みます。Java言語はまさにこのような形で処理系が構成されています(javacがコンパイラ、javaが仮想マシンのプログラム)。

言語処理系の構造

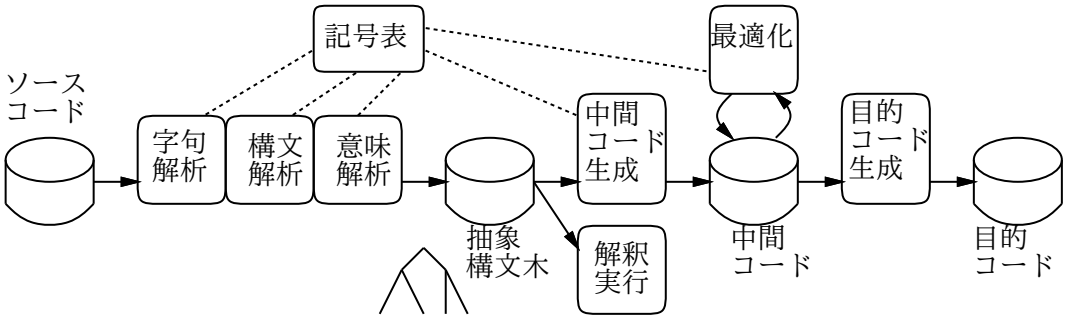

図 1: 言語処理系の一般的な構成

一般的な言語処理系の大まかな構造を、図??に示します。ディスク記号で記したのがデータ構造、四角で囲んだのが処理系内部のコンポーネントです。

コンパイラの場合、その要素は大きくわけてソースコードから情報を取り出す解析部 (**analysis phase**) ないしフロントエンド (**front end**) と、その情報に基づいて目的コードを生成していく生成部 (**synthesis phase**) ないしバックエンド (**back end**) から成ります。図??でいって、中央にある「抽象構文木」の左側が解析部、右側が生成部になります。なお、ここで示しているのはあくまでも一般的な構成であり、処理系によってはこれと異なる形になることもあります。

各部分の詳細についてはこれから時間をかけて解説していきますが、ここでは各部分の大まかな機能や構造について説明します。

言語処理系の構造 (2)

ソースコード (source code) — プログラミング言語の形で記述されたプログラム。言語処理系の入力となる。

字句解析 (lexcal analysis) — ソースコードを「名前」「定数」「記号」などのかたまり (トークン) に分割する。コメントの削除などの処理もここで行なう。

構文解析 (syntax analysis) — トークンの列に対して構文規則のあてはめを行ない、「メソッド」「文」「式」などの要素がどの範囲でどういう構造になっているかを決定する。

意味解析 (semantic analysis) — 関数、変数、型などの使用状況を解析し、名前の未定義や多重定義、型の不一致などの誤りを検出する。

記号表 (symbol table) — これは受動的なデータ構造である場合もある。コンパイラの各コンポーネントは名前、型などの情報を記録し、また必要に応じて参照する必要があるが、それらの情報を保持し必要なサポートを行なうのが記号表である。

抽象構文木 (abstract syntax tree) — ソースコードの情報から後の段に必要な構造や情報を抽出した結果は木構造のデータで表現することが多く、これを抽象構文木と呼ぶ。

言語処理系の構造 (3)

解釈実行 (interpritive execution) — インタプリタの場合は、抽象構文木から直接実行することもでき、簡易な処理系で多くこの形が使われる。ただし実行速度が遅くなるので、より高速にしたい場合は仮想マシン型の構成が使われるので、コード生成の側に進む。

中間コード生成 (intermediate code generation) — 目的コード (機械語やアセンブリ言語) は内部処理には向いていないので、いったん中間的な形式のコードを生成し、最適化に進むことが普通である。簡易な処理系で最適化を最小限にしたものでは、抽象構文木から直接目的コード生成につながるものもある。

中間コード (intermediate code) — 最適化の処理に適した形でプログラムの動作を表現するコード形式。

最適化 (optimization) — プログラムの中の無駄を削減したり、意味を変えない範囲でより高速に実行できる形に書き換えるなどして、目的コードの実行が速くなるようにする。ここでは中間コードに対してさまざまな最適化を繰り返し行なうイメージで描いてあるが、そのほかに目的コード生成時に行なう最適化もある。

言語処理系の構造 (4)

目的コード生成 (**target code generation**) — 中間コードから目的コードの形式に変換する。目的コードが機械語やアセンブリ言語など低水準 (**CPU** 依存) 形式の場合、他の **CPU** 向けにコンパイラを移植するときはこの部分を変更する。

目的コード (**target code**) — 最終的に出力されるコードで、機械語やアセンブリ言語のこともあるが、仮想マシン方式のように仮想マシン語を出力したり、他の高水準言語のソースコードを出力することもある。

抽象構文木とその利用

抽象構文木

抽象構文木 (**abstract syntax tree, AST**) は既に学んだ構文木に基づいていますが、構文木の「文法に正確に対応する」という制約は無くし、また言語処理系にとって必要な情報を適宜データ構造 (具体的には木のノード) に追加するという形で作られています。

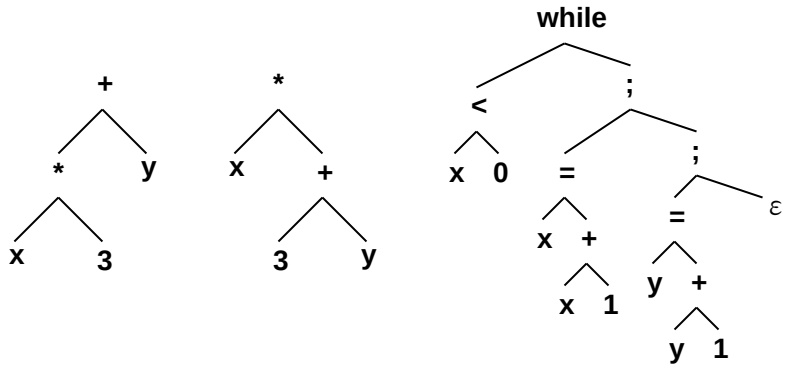

図 2: 抽象構文木の図解例

図??に抽象構文木を図解した例を示します。このように、抽象構文木ではノードのところにそのノードに対応する意味づけ (演算の種類など) を書くことが多いです。図??左と中はそれぞれ「`x * 3 + y`」と「`x * (3 + y)`」に対応する木構造を表しています。また、図??右は

$$\textbf{while (x ¡ 0) \{ x = x + 1; y = y + 1; \}}$$

に対応する木構造を表しています。このように、文の並びは「並びを表すノード」を使って数珠つなぎにして表現することが普通です。いずれの場合も、「(⋯)」や「{ ⋯ }」などの構造を指定するための記法は木構造の上では無くなっていることに注意 (木構造自体でその構造が表せるためにそうなっています)。

Javaによる式木の実装

式木 (**expression tree**) とは、式を表現する抽象構文木を言います。ここでは整数のみの式を扱い、「変数」「定数」「加算」「乗算」のノードを作ることにしました。まず冒頭とmain()の部分だけを示します。

```
import java.util.*;

public class Sam31 {
  static Map<String,Integer> vars = new TreeMap<String,Intege

  public static void main(String[] args) {
    vars.put("x", 5);
    Node expr = new Add(new Mul(new Var("x"), new Lit(3)), ne
    System.out.println(expr);
    System.out.println(expr.eval());
  }
  // ここに他のstaticな内部クラスを入れる
}
```

Map<T_1,T_2>は型T_1をキーとし、型T_2を値として保持するような表を表す型です (**Java**の用語としてはインタフェース)。ここでは、文字列 (=変数名) をキーとし、その変数の値 (=整数) を保持する型Map<String,Integer>のstatic変数varsを用意し、そこにTreeMap<String,Integer>のインスタンスを作って格納します。TreeMapは赤黒木 (という名前のデータ構造で、キーの昇順に項目が並べられる) を用いた表の実装です。

このように、**Java**では変数にはその型と互換性のあるさまざまな型のオブジェクトを格納することができます。互換性の規則についてはまた後で説明します。

Javaによる式木の実装 (2)

そして、この変数varsはstaticなので、後で出て来るノードのためのクラスもすべてこのクラスのstaticな内部クラスとすることでどこからでも参照できるようにしています。

そしてようやく、main()の動作ですが、まず冒頭で、vars の"x"の項目に5を入れます。これは変数xに5 が入っていることを表しているつもりです。その次に、様々なノードのコンストラクタを使って式木を組み立てます。すべてのノードはNodeと互換性があるように作ってあります。ここで組み立てている式は「tt (x * 3) + 1」に相当します。

その後、まず作成したオブジェクトを文字列表示し、続いてeval()により値を計算して表示します。main()の後には多くのクラスが出て来ますが、その説明の前に実行例を示そう。xには5を入れてあるので、確かに正しい値が計算できています。

```
% java Sam31
((x*3)+1)
16
%
```

式木のノード群と継承

では、式木のノードに対応するクラス群を読んでいくことにします。これらはすべて、クラス `Sam31` の内部クラスで、かつ `static` です。

ここで新たな概念として、継承 (**inheritance**) について説明します。継承とは、複数の類似した (関連性のある) クラスを作るときに「あるクラス C を土台として別のクラス D を作る」操作です。このとき、C を基底クラス (**base class**) ないし親クラス (**parent class**)、D を派生クラス (**derived class**) ないし子クラス (**child class**) と呼びます。継承をおこなうには、子クラス D の冒頭に「`extends` C」という指定をおこないます。

継承をおこなうと、次の **3** つの効果が得られます。

1. クラス C で定義されている変数やメソッドの定義はそのまま D にも引き継がれる (コピーされてくると考えればよい)。
2. メソッド定義については、同じパラメタや返値をもつ同名のメソッドを再定義することで、継承してきたメソッドをオーバライド (**override**、上書きの意味) できる。
3. 型 D の値は型 C に互換 (**compatible**) となる。つまり型 C の変数にクラス D のインスタンスを入れられる。

これらに加え、クラス D で新たなインスタンス変数やメソッドを追加することは自由です。

上記の事項のうち **3.** については、D は C からメソッドやデータ構造を引き継いでいるので、D のインスタンスはほぼ C のインスタンスと互換性があるようになるので、こうなっています (実際にはオーバライドのしかたや機能に追加により互換な動作ができないこともあり得ますが、なるべくそうはしないというのがお約束です)。

式木のノード群と継承 (2)

また、D を親としてさらに継承した E を作ることもあり、その場合はこれらのクラスもすべて C と互換性があることになります。実は **Java** ではとくに指定のないクラスは `extend Object` を指定したとして扱われるので、すべてのクラスは `Object` に互換であり、このクラスから必要最小限の機能を継承しています。

継承の機能の話題に戻って、事項の **1.** のおかけで、類似した機能を持つ多数のクラスを少ない行数で書くことができます。それは D を定義するときに、C から基本は引き継いできて、ことなる部分だけ記述すればよいからです。これを差分プログラミング (**differential programming**) と呼びます。事項 **2.** については、継承してきたそのままでは機能が十分でないことがあることによります。

また、クラス群の設計上、必ずオーバライドが必要なメソッドを定義することもあります。たとえば、今回の例ではクラス `Node` がすべてのノードの基底となり、そこで定義するメソッド `eval()` は「値を計算する」機能となりますが、実際の計算方法は「加算」「変数」など個別のクラスでなければ決められません。そこで、`Node` ではメソッド `eval()` を抽象メソッド (**abstract method**) と指定し (コード定義を持たない)、その子孫のクラスでオーバライドすることを指定します。抽象メソッドを持つようなクラスは抽象クラス (**abstract class**) と呼ばれ、インスタンスを作ることはできません。

式木のノード群と継承 (3)

では実際に、抽象クラス Node を見てみます。

```
abstract static class Node {
  List<Node> child = new ArrayList<Node>();
  public void add(Node n) { child.add(n); }
  public abstract int eval();
}
```

インスタンス変数 child はノードの並びを保持する List<Node> 型の変数であり、その実装としては配列のように機能する ArrayList<Node> のインスタンスを入れます。メソッド add はそこに子ノードを追加するメソッドです。そしてメソッド eval() は前述にように抽象メソッドであり、コードを持ちません。

式木のノード群と継承 (4)

次に、このクラスを継承して作る定数と変数のクラスを見てみます。

```
static class Lit extends Node {
  int val;
  public Lit(int v) { val = v; }
  public int eval() { return val; }
  public String toString() { return ""+val; }
}
static class Var extends Node {
  String name;
  public Var(String n) { name = n; }
  public int eval() { return vars.get(name); }
  public String toString() { return name; }
}
```

定数の方は整数の値をインスタンス変数 val に保持します。その値は生成時にコンストラクタで受け取り代入します。eval() はその値を返せばよいです。なお、toString() は Object から継承しているメソッドであり、そのインスタンスを打ち出すなどのときに文字列に変換する際に呼び出されるます。ここでは val に値を空文字列と連結して得られる文字列オブジェクトを返します。[1]

変数では、名前の文字列を保持する name と値を格納する表 (Map<String, を保持する tbl の 2 つがインスタンス変数であり、いずれもコンストラクタで受け取って初期設定します。eval() はその表から変数名を指定して値を取り出して返します。toString() は名前の文字列を返せば良いです。

[1] Java では文字列と何かを連結するとその「何か」が文字列に自動変換されてから連結されることになっている。

式木のノード群と継承 (5)

次は2項演算である加算と乗算ですが、これらは共通する内容があるので、その共通部分を BinOp という抽象クラスにまとめました。

```
abstract static class BinOp extends Node {
  String op;
  public BinOp(String o, Node n1, Node n2) { op = o; add(n1
  public String toString() { return "("+child.get(0)+op+chi
}
```

具体的には、演算を表す文字列 op をインスタンス変数として追加し、コンストラクタでその文字列と2つのノード (演算される2つの被演算式に対応) を受け取り、op は初期化し、2つのノードは child(継承してきているインスタンス変数で、List<node> を保持) に順次追加しています。toString() では、2つの被演算式を中央に演算子いを入れて連結し、全体を「(...)」で囲んだ文字列を返しています。

式木のノード群と継承 (6)

以上があればAddとMulは簡単で、eval()で2つの被演算式を計算してそれらを加算/乗算して返すようにするだけです。ただしあと1つ、自らのコンストラクタの中で初期化のためにBinOpのコンストラクタを呼び出さなければなりません。Javaでは子クラスのコンストラクタから親クラスのコンストラクタを呼び出すときには「super(…)」という書き方で呼び出すことになっています。

```
static class Add extends BinOp {
  public Add(Node n1, Node n2) { super("+", n1, n2); }
  public int eval() { return child.get(0).eval() + child.ge
}

static class Mul extends BinOp {
  public Mul(Node n1, Node n2) { super("*", n1, n2); }
  public int eval() { return child.get(0).eval() * child.ge
}
```

式木のノード群と継承 (7)

演習 3-1 上の例題のコードを入手してそのまま動かせ。動いたら、次のことをやってみょ。ノードを追加したら正しく動作することを確認すること。

a. 別の計算式を構成し、その計算ができることを確認する。

b. 減算、除算、剰余のノード Sub、Div、Mod を追加する。

c. 比較演算子のノード Eq、Ne、Gt、Ge、Lt、Le を追加する。いずれも、条件が真なら「1」、偽なら「0」を値とする。

d. && に相当する And、|| に相当する Or、! に相当する Not を追加する。Not は被演算子が 1 つなので多少工夫が必要。

e. 代入のノード Assign を追加する。コンストラクタは変数のノードと一般の式のノードを受け取る。exec() では式を計算し、その値を変数表に格納するとともに、全体の値としては代入した値を返す。

f. 順次実行のノード Seq を追加する。動作としては 2 つのノードを保持し、それらを順に実行し、値としては 2 つ目のものの値を返す。2 つ目のノードは nil でもよく、その場合は 2 つ目は実行せずに 1 つ目の値を返す。

より複雑なノード

とりあえず、先の演習問題関係のノードを示します。

```
static class Sub extends BinOp {
  public Sub(Node n1, Node n2) { super("-", n1, n2); }
  public int eval() { return child.get(0).eval() - child.ge
}
static class Div extends BinOp {
  public Div(Node n1, Node n2) { super("/", n1, n2); }
  public int eval() { return child.get(0).eval() / child.ge
}
static class Mod extends BinOp {
  public Mod(Node n1, Node n2) { super("%", n1, n2); }
  public int eval() { return child.get(0).eval() % child.ge
}
static class Eq extends BinOp {
  public Eq(Node n1, Node n2) { super("==", n1, n2); }
  public int eval() { return child.get(0).eval()==child.get
}
static class Ne extends BinOp {
  public Ne(Node n1, Node n2) { super("!=", n1, n2); }
  public int eval() { return child.get(0).eval()!=child.get
}
```

より複雑なノード (2)

```
static class Gt extends BinOp {
  public Gt(Node n1, Node n2) { super(">", n1, n2); }
  public int eval() { return child.get(0).eval()>child.get(
}
static class Ge extends BinOp {
  public Ge(Node n1, Node n2) { super(">=", n1, n2); }
  public int eval() { return child.get(0).eval()>=child.get
}
static class Lt extends BinOp {
  public Lt(Node n1, Node n2) { super("<", n1, n2); }
  public int eval() { return child.get(0).eval()<child.get(
}
static class Le extends BinOp {
  public Le(Node n1, Node n2) { super("<=", n1, n2); }
  public int eval() { return child.get(0).eval()<=child.get
}
```

ここまでは普通でしたが、And と Or は多少の工夫が必要です。それは、And であれば、左辺が 0(false) だったら、もう右辺を評価する必要はない、ということです (Or も同様)。このことが分かるように if 文を使って実現しています。Not についてはそのような複雑さはありませんが、せっかく BinOp を作ったので被演算子が 1 個の場合用の UniOp も作りました。

より複雑なノード (3)

```
static class And extends BinOp {
  public And(Node n1, Node n2) { super("&&", n1, n2); }
  public int eval() {
    int v = child.get(0).eval();
    if(v == 0) { return 0; } else { return child.get(1).eva
  }
}
static class Or extends BinOp {
  public Or(Node n1, Node n2) { super("||", n1, n2); }
  public int eval() {
    int v = child.get(0).eval();
    if(v != 0) { return v; } else { return child.get(1).eva
  }
}
abstract static class UniOp extends Node {
  String op;
  public UniOp(String o, Node n1) { op = o; add(n1); }
  public String toString() { return "("+op+child.get(0)+")"
}
static class Not extends UniOp {
  public Not(Node n1) { super("!", n1); }
  public int eval() { return child.get(0).eval()==0?1:0; }
}
```

より複雑なノード (4)

代入ですが、右辺を評価して値を得るところは他の演算と同様で、その後値を vars に格納する必要があります。それには変数名が必要ですが、Var オブジェクトは toString() を呼べば名前の文字列が返されるのでそれを使っています。

```
static class Assign extends Node {
  Var v1; Node n1;
  public Assign(Var v, Node n) { v1 = v; n1 = n; }
  public int eval() {
    int v = n1.eval(); vars.put(v1.toString(), v); return v
  }
  public String toString() { return v1+"="+n1; }
}
```

より複雑なノード (5)

Seq ですが、問題に説明したものより使いやすくするため、コンストラクタでは Node を可変引数で受け取ることにしました。つまり Node 型の値をいくつでもパラメタとして書くことができ、受け取る側では 1 つの配列として受け取ります。内部では受け取った配列の各要素を順次 child に add() しています。この「for(変数: 式) …」というのは **foreach** ループと呼ばれ、「式」は複数の値を順次返すオブジェクト (イテレータ) を生成できるものである必要がある (配列はそのようにできています。また List<T>もそうである)。eval() では各ノードを順次実行して最後の値を返します。toString() では並びを中かっこで囲んで、各要素ごとに改行文字をつける (文を並べるのに使うことが多いので、その方が読みやすそうです)。

```
static class Seq extends Node {
  public Seq(Node... a) { for(Node n:a) { child.add(n); } }
  public int eval() {
    int v = 0;
    for(Node n:child) { v = n.eval(); }
    return v;
  }
  public String toString() {
    String s = "{\n";
    for(Node n:child) { s += n.toString() + ";\n"; }
    return s + "}";
  }
}
```

より複雑なノード (6)

次はRead とPrint です。に前者はコンストラクタで指定した変数の名前を含む文字列をプロンプトとして表示した後、整数を入力し、その値を変数に入れた上で結果としてもその値を返します。後者は式を1つ持ち、その式の評価結果を出力します。

```
static class Read extends Node {
  Var v1;
  public Read(Var v) { v1 = v; }
  public int eval() {
    System.out.print(v1+"? ");
    Scanner sc = new Scanner(System.in);
    String str = sc.nextLine();
    int i = Integer.parseInt(str);
    vars.put(v1.toString(), i); return i;
  }
  public String toString() { return "read "+v1; }
}
static class Print extends Node {
  public Print(Node n1) { child.add(n1); }
  public int eval() {
    int v = child.get(0).eval(); System.out.println(v); ret
  }
  public String toString() { return "print "+child.get(0);
}
```

より複雑なノード (7)

最後はwhile文とelseのないif文です。どちらも、中では結局Javaの同じ構文を使って実現しています。

```
  static class While extends Node {
    public While(Node n1, Node n2) { child.add(n1); child.add
    public int eval() {
      int v = 0;
      while(child.get(0).eval() != 0) { v = child.get(1).eval
      return v;
    }
    public String toString() {
      return "while("+child.get(0)+")"+child.get(1);
    }
  }
  static class If1 extends Node {
    public If1(Node n1, Node n2) { child.add(n1); child.add(n
    public int eval() {
      int v = 0;
      if(child.get(0).eval() != 0) { v = child.get(1).eval();
      return v;
    }
    public String toString() { return "if("+child.get(0)+")"+
  }
}
```

より複雑なノード (8)

では、これらを使ってそれらしいプログラムを組み立てて動かしてみましょう。次は N を入力すると 1 から N までの階乗を順次打ち出すプログラムを (抽象構文木として) 組み立て、実行しています。

```
import java.util.*;

public class Sam32 {
  static Map<String,Integer> vars = new TreeMap<String,Intege

  public static void main(String[] args) {
    Node prog = new Seq(
      new Read(new Var("x")),
      new Assign(new Var("i"), new Lit(0)),
      new Assign(new Var("p"), new Lit(1)),
      new While(new Lt(new Var("i"), new Var("x")),
        new Seq(
          new Assign(new Var("i"), new Add(new Var("i"), new
          new Assign(new Var("p"), new Mul(new Var("p"), new
          new Print(new Var("p"))))));
    System.out.println(prog);
    System.out.println(prog.eval());
  }
  // ここに他のstaticな内部クラスを入れる
}
```

より複雑なノード (9)

実行例は次のようになります。

```
% java Sam32
{              ←ASTの表示
read x;
i=0;
p=1;
while((i<x)){
i=(i+1);
p=(p+i);
print p;
};
}
x? 5       ←Readによる入力
1          ←順次階乗を出力
2
6
24
120
120        ←最終的な値も120
%
```

より複雑なノード (10)

演習 3-2 上記のノードを用いて次のようなプログラムを (抽象構文木として) 組み立てて動かしてみよ。

a. N を入力し、N 以下のフィボナッチ数を出力。
b. N を入力し、N の因数をすべて出力。
c. N を入力し、N 以下の素数を出力。
d. N を入力し、N の 2 進表現に「1」のビットが何個あるか出力。
e. その他、自分でやってみたいと思う計算をする。

演習 3-3 上記のノード群に、さらに次のような機能を持つノードを追加してみよ。プログラムを (抽象構文木で) 組み立てて動作を確認すること。

a. C や Java の do-while ループを実現するノード `DoWhile`(ループ本体のノード，条件式のノード)。
b. else 部のある if 文を実現するノード `If2`(条件のノード，*then* 部のノード，*else* 部のノード)。
c. 回数を指定してループ本体をその回数だけ繰り返すノード `Times`(回数の式のノード，ループ本体のノード)。
d. 上記と同様だが、ただし周回ごとに指定した変数が $0, 1, 2, \cdots$ と変化していくノード `TimesFor`(回数の式のノード、変数のノード、ループ本体のノード)。
e. 式を指定してその式の値に応じてどれかの選択肢を実行するノード `Switch`(式のノード，`new Node[]`$\{B_1, B_2, \ldots, B_n\}$)。式の値が 1 なら B_1, 2 なら B_2, 等が実行される (どれでもない場合はどれも実行されない)。
f. 自分で作ってみたいと思うノードを構想し作ってみよ。

課題 3A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題＃3」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. コンパイラの構成についてどれくらい知っていましたか。また、どの部分にとくに興味がありますか。

  Q2. 抽象構文木でプログラムを組み立ててみてどのように思いましたか。

  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。