オブジェクト指向技術'11 ＃3 — 言語処理系とOO

久野　靖*

2011.4.28

1　ソフトウェアアーキテクチャと言語

アーキテクチャとは何でしょうか? もともと建築 (物) などに使う用語ですが…

> 「作り上げることの芸術ないし科学。特に、美的要因と実用的要因の両方を考慮した上で、人が使うための建築物をデザインし作り上げる技のこと。」
>
> Shorter Oxford Dictionary 5th ed.
> 「ビューティフルアーキテクチャ」の「はじめに」参照

以下では、ソフトウェアの場合について考えます。上の引用からすると、アーキテクチャとは「美しい設計」ということでいいのでしょうか? それだけではないように思えます。つまり、ソフトウェアアーキテクチャとは — ソフトウェアの『構造の構造』だと思うわけです。具体的にこれがどういうことかというと。

中規模までのソフトウェアでは、「構造を設計」してその構造を作ればよいのですが、実際のソフトウェアには変更がつきものです。そして、その変更が最初に作った構造にうまく適合しないと (または変更を及ぼす人がヘボだと)、どんどん構造が汚くなってメンテナンスできなくなりますね (よく知られている通り)。

そこで、「構造がどのようであるべきか」ということをまず設計して、その設計にしたがって「具体的な構造を決める」ことを行う、というふうに 2 段階にするわけです。これが「アーキテクチャ + その実装」ということになります。なぜこれがよいかというと、「どのようであるべきか」が明示されているので、直すときもその「あるべき」に従うことで、既存の部分と整合した形が維持できるからです (図 1)。

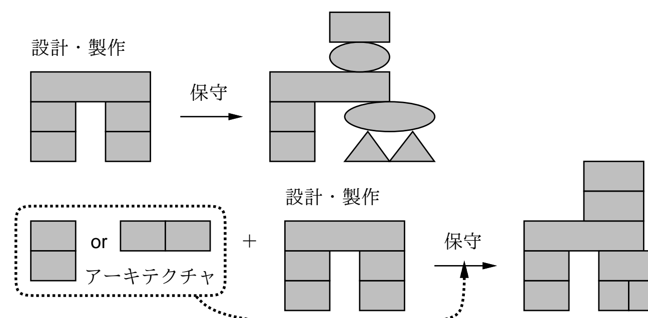

図 1: ソフトウェアアーキテクチャ

そして、柔軟性が高く、また変更 (成長/改良) が容易なアーキテクチャは「言語」を中心としたものとなることが多くあります。これは、アーキテクチャが持つ「規則・制約」を、全部人間が読み取っ

---

*経営システム科学専攻

てそれに従いながら設計・構築するよりも、その規則や制約に基づいた構造に従う部分は「おまかせ」にしつつ、「その上で何をどのようにしたいか」だけを記述することで実際にアーキテクチャに従ったものを「生成」できる方が、記述量の点でもアーキテクチャへの準拠性の点でも変更の容易さの点でも有利だからです。

複雑かつ拡張可能なソフトウェアを言語をコアとしてとりまとめた古典的な例としてはGNU Emacsがあります。GNU EmacsはEmacs Lisp言語の処理形+APIというコアとそれによって記述されたエディタパッケージという2つのレイヤに分かれていますが、これによってシステムを壊したりクラッシュさせたりすることなく、誰もが簡単に(Emacsの流儀に従った) Lispコードを書いてロードすることで、自分流にエディタをカスタマイズしたり、自分が欲しい機能を持ったエディタを作り出したりできるわけです。

せっかく話が出たので、Emacs Lispで簡単なコマンドを作ってみましょう。次の内容をたとえば`test1.el`というファイルに入れます。

```
(defun mult-indent (arg)
  "multiply indent by argument"
  (interactive "p")
  (beginning-of-line)
  (if (equal arg 1) (setq arg 2))
  (while (< (point) (point-max))
    (let ((count (del-space-count)))
       (insert-char 32 (* count arg)))
    (forward-line 1)))
(defun del-space-count ()
  (let ((count 0))
    (while (char-equal (following-char) 32)
       (delete-char 1)
       (setq count (1+ count)))
    count))
(global-set-key "\C-xz" 'mult-indent)
```

作成しているコマンドは「字下げをN倍にする」もので、特に指定しなかった場合は2倍とします。関数`mult-indent`が作成するコマンドの関数で、「`(interactive "p")`」というのは、この関数がコマンドとして呼び出せ、かつ数値引数を取ることを表します。まず行の先頭まで行き、引数が1のときは2にし(2倍)、次にバッファの最後にくるまで繰り返し、行頭の空白を数えながら消して、その数の引数倍の空白を挿入し、次の行に進みます。`del-space-count`は文字通り、空白を削除してその空白数を返します。最後に`global-set-key`で`mult-indent`を「Control-X z」にバインドします。これを動かす時は

```
Control-U 3 Control-X z
```

とするとたとえば3倍になるわけです(引数を指定しないと2倍)。

演習1 このプログラムをロードして動かしてみなさい。納得したら、バッファに対して別のことをするコマンドに直してみなさい。

2 言語の構文とその処理

2.1 BNFによる構文記述と構文木

Emacs Lispの場合はLispという汎用の言語ですから、言語そのものがソフトウェア(エディタ)に対して及ぼす限定(構造化)というのはほぼなく、単に提供されているAPIや処理系としての制約

(バッファに対して行える操作が限定されている、バッファの表示は勝手に行われるなど) しかありませんでした。

これに対し、もっと記述対象のソフトウェアに特化した構文や構文に付随する制約を設けることで、対象をコンパクトに記述できたり、記述の間違いを初期の段階で (コンパイル時に) 発見して指摘できたりするようになります。今日では、特定ドメインに特化したコンピュータ言語のことを DSL(Domain Specific Language) と呼び、用途によってさまざまなものが使われています。以下では DSL などにも使える、一般的な言語処理系の技術を簡単に取り上げます (詳しくやると通年の講義くらいの内容はあるので簡単に)。

まず最初に、BNF から取り上げます。言語ではその書き方 (構文) を文脈自由文法で規定するのが普通で、さらに記法としては BNF(Backus-Naur Form、Backus や Naur はこの記号を使い始めた人の名前) で記述することが普通です。BNF は

    記号 ::= 記号 記号 ⋯

という形の定義を連ねるだけで (つまり非常に基本的な書き方だけで) 文脈自由文法を記述できることが特徴です (少し書き方をコンパクトにするために、「または」を表す「|」を使うこともありますが)。記号には、次の 2 種類があります。

- 非端記号 — 言語の構文上の概念を説明するために導入される記号であり、BNF の左辺に現れる (文法によって定義される)
- 端記号 — プログラムの字面上に現れる単位 (識別子、リテラル、予約後など) に対応する記号であり、BNF の左辺には現れない (文法内では定義されない)

たとえば、「数値をコンマで区切った並び」を表してみましょう。

    並び ::= 数値 | 数値 "," 並び

なお、この「または」は次の 2 つの定義の並置と同等です。

    並び ::= 数値
    並び ::= 数値 "," 並び

ここで「並び」は非端記号、「数値」と「","」が端記号になります。数値はここでは単に「数字の並び」であることにします。そうすると、

```
100
11, 12, 23
```

などはこの「並び」にあてはまり、

```
100,
11, 12,, 23
11 12
```

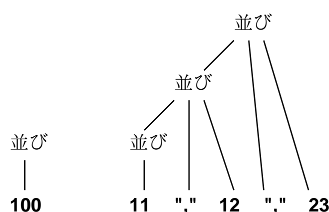

図 2: 「並び」の構文木

などはあてはまらないと分かるわけです。そして、このような規則に従っているかどうかを調べるコードを書くのは結構大変ですが、BNF(ないし文脈自由文法) で記述してあれば、ツール (後述) によってあてはまりを処理するコードが生成できるという利点があります。なお、あてはまりの様子を示すのに、図 2 のような、文法に対応した形の木構造を描くことがよくあります。これを構文木 (syntax tree) と呼びます。

演習 2 次のものを表す BNF を書いてみなさい。

a. 数値と引き算のみから成る式 (端記号として「数値」「"-"」を使うものとする)。
b. Lisp の S 式 (端記号として「名前」「数値」「"("」「")"」を使うものとする)。
c. 四則演算のみから成る数式 (端記号として「名前」「数値」「"("」「")"」「"+"」「"-"」「"*"」「"/"」を使うものとする)。

さらに、その BNF に対応する、次のものの構文木を描いてみなさい。

a. 「`10 - 8 - 3`」
b. 「`(list (list 1 2) 3)`」
c. 「`3 / (2 * a + 5)`」

この演習からも分かることですが、構文を定義するにあたっては、「正しい形のものを受け付ける」だけでなく、「プログラムの構造に対応した構文木ができるようにする」ことも大切です。また、演算子の順位のようなものを扱うと、A → B、B → C、C → D のように 1 本につながった枝が沢山できます。処理の上でわずらわしいので、このような枝を省略して扱うこともよくあります。これを抽象構文木 (Abstract Syntax Tree, AST) と呼びます。

2.2 抽象構文木のデータ構造

では実際に、抽象構文木をコードで実現してみます。ここでは簡単に構造が打ち込めるということから Ruby を採用します。まず `Node` という基底クラスを作って、初期化と表示の機能を持たせます。基本的に左右 2 つの子供`@left`、`@right` と「自分が何のノードか」を表す`@op` がインスタンス変数です。

```
class Node
  def initialize(l=nil, r=nil)
   @left = l; @right = r; @op = '?'
  end
  def to_s()
    return '(' + @left.to_s + @op.to_s + @right.to_s + ')'
  end
end
```

続いて、四則のノードを作り、それぞれの初期化と「実行」を定義します。

```
class Add < Node
  def initialize(l, r) super; @op = '+' end
  def exec() return @left.exec + @right.exec end
end
class Sub < Node
  def initialize(l, r) super; @op = '-' end
  def exec() return @left.exec - @right.exec end
```

```
  end
  class Mul < Node
    def initialize(l, r) super; @op = '*' end
    def exec() return @left.exec * @right.exec end
  end
  class Div < Node
    def initialize(l, r) super; @op = '/' end
    def exec() return @left.exec / @right.exec end
  end
```

あと、「葉」のノードは「整数 (リテラル)」と「変数」の 2 つを用意します。リテラルの値は`@left` にそのまま入れ、変数は`@left` に名前を入れておいてハッシュ`$vars` に対応する値を格納するようにします。

```
  class Lit < Node
    def exec() return @left end
    def to_s() return @left.to_s end
  end
  $vars = {}
  class Var < Node
    def exec() return $vars[@left] end
    def to_s() return @left.to_s end
  end
```

これを読み込んで動かすようすは次の通り。

```
  % irb
  irb> load 'tree1.rb'
  => true
  irb> $vars['x'] = 3
  => 3
  irb> tree = Add.new(Mul.new(Var.new('x'),Lit.new(3)),Lit.new(5))
  => ...
  irb> tree.to_s
  => "((x*3)+5)"
  irb> tree.exec
  => 14
```

このように、ノードをオブジェクトに対応させることで、素直に抽象構文木が表現でき、式の評価ができるわけです。

式と書きましたが、抽象構文木はもちろん、式にとどまるわけではありません。もう少しノードの種類を増やしてみます。具体的には、「右辺の値を左辺の変数に代入する」ノード、「決まった回数だけループする」ノード、「2 つのノードを順番に実行する」ノードです (あと「何もしない」ノードも一応用意しました)。

```
  class Assign < Node
    def initialize(l, r) super; @op = '=' end
    def exec() v = @right.exec; $vars[@left.to_s] = v; return v end
  end
  class Seq < Node
```

```
    def initialize(l, r) super; @op = ';' end
    def exec() @left.exec; return @right.exec end
  end
  class Loop < Node
    def initialize(l, r) super; @op = 'L' end
    def exec()
      v=0; @left.exec.times do v=@right.exec end; return v
    end
  end
  class Noop < Node
    def exec() return 0 end
    def to_s() return '?' end
  end
```

これを用いて、「5 の階乗」を計算するプログラムを組み立てて実行してみます。

```
  def test1
    e =
      Seq.new(
        Assign.new(Var.new('n'), Lit.new(5)),
        Seq.new(
          Assign.new(Var.new('x'), Lit.new(1)),
          Seq.new(
            Loop.new(
              Var.new('n'),
              Seq.new(
                Assign.new(Var.new('x'), Mul.new(Var.new('x'),
                           Var.new('n'))),
                Assign.new(Var.new('n'), Sub.new(Var.new('n'),
                           Lit.new(1))))),
            Var.new('x'))))
    puts(e)
    return e.exec
  end
```

実行のようすは次のとおり。

```
  irb> test1
  ((n=5);((x=1);((nL((x=(x*n));(n=(n-1))));x)))
  => 120
```

このように、木構造を直接実行する形の言語処理系を「ツリーインタプリタ」と呼びます。ツリーインタプリタは比較的簡単に作れるので、現在でも速度があまり問題にならないような言語では多く使われています。なお、上の例では言語のソースコードから木構造を作る部分がありませんでしたが、そこは少し後で扱います。

演習 3 このノード構造を使って、次のプログラムを組み立てて実行してみなさい。

a. 2^N を計算する。

b. N 番目のフィボナッチ数を計算する。

c. ${}_nC_r$ を計算する。

3　Visitor パターン

前節で取り上げたオブジェクトの木構造を見て「すばらしい」「分かりやすい」と思いましたか? もしそうなら、何か見落としています (笑)。何が問題かというと、「ツリーを実行する」ためのメソッドが各クラスにバラバラに散らばっているという点です。たとえば、式の計算について見ると、加算、減算などの処理はすべて `Add`、`Sub` など各クラスのメソッドとして書かれています (図 3)。

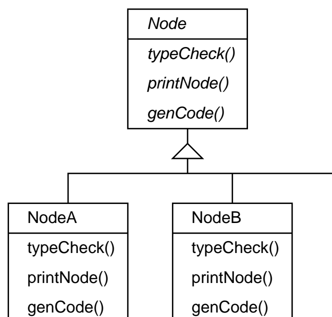

図 3: 素朴な再帰による木構造の処理

それで別にいいじゃないか、と思いますか? これには、次のような問題点があります。

- 「計算をする」というひとまとまりの処理が各クラスにバラバラに散らばってしまう。
- 「計算する」「型チェックする」「コード生成する」などの処理ごとにそのバラバラのクラスに全部手を入れてメソッドを追加しなければならない。
- それぞれのクラスでやる処理全体を通して使うデータの置き場所がない (子供メソッドの呼び出しごとにデータ構造を持ち回るのも繁雑)。

この問題を解消するために考案されたのが Visitor パターンです (図 4)。こんどは、各ノードは `NodeVisitor` オブジェクト `v` を引数として受け取る `accept()` というメソッドだけを持っていて、その中で「`v.visit` ノード種別 `()`」というメソッドを呼び出すことで「自分の種別の処理」を呼び出します。

`NodeVisitor` はインタフェースであり、型検査、コード生成などの仕事ごとにそれを実装するさまざまな `Visitor` オブジェクトをこのインタフェースに従うものとして用意します。いずれかの `Visitor` オブジェクトを引数として渡して `accept()` を呼び出すと、ノードの種類ごとに `visit` ノード種別 `()` が呼ばれて来ますから、その中で子ノードをさらにたどることが必要なら `accept(this)` を呼び出せばつぎつぎにだどりが行えます。

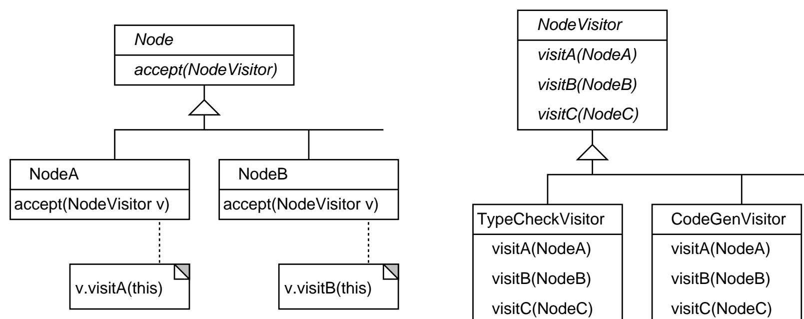

図 4: Visitor パターンへの変換

なお、これらの呼び出しはすべて 1 つの `Visitor` オブジェクトのメソッド呼び出しなので、そのオブジェクトのインスタンス変数が「ずっと保持しておくデータの置き場所」に使えます。

さて、ここまでがガンマ本 (デザパタ本) に載っている Visitor の話で、Visitor には元の形式の弱点を解消した代償として次の弱点が生じていると書かれています。

> オブジェクト構造のクラスをしばしば変更する場合には、すべての Visitor のインタフェースを再定義する必要があり、潜在的にコストは高くつく。

確かにそれはその通りですが、だからといって元の形に戻るのも面白くありません。そこですぐ後で出て来る SableCC は、次のように Visitor パターンをさらに拡張しています。

- 切り替えを行うインタフェースを非常に抽象的なものとして (何も操作なしで) 定義する。

```
interface Visitor { }
```

- 各ノード側はこれを受け取る `accept()` を定義するため、次のインタフェースを実装する。

```
interface Visitable { void apply(Visitor v); }
```

- 実際の Visitor は上のインタフェースを拡張して個々の種別を定義。

```
interface BaseVisitor extends Visitor {
  void visitA(A obj);
  void visitB(B obj);
  ...
}
```

- 別の種別も増やすことができる。

```
interface ExcendedVisitor extends Visitor {
  void visitX(X obj);
  void visitY(Y obj);
  ...
}
```

- 個々のノードの `accept()` は自分が属しているインタフェースにキャストして自分用のメソッドを呼び出すことができる。

```
class NodeA extends implements Visitable {
  ...
  void accept(Visitor v) { ((BaseVisitor)v).visitA(this); }
}
class NodeX extends implements Visitable {
  ...
  void accept(Visitor v) { ((ExtendedVisitor)v).visitX(this); }
}
```

- 実際に使うときには、これらをすべてまぜたインタフェースを作る。

```
interface AllVisitor extends BaseVisitor, ExtendedVisitor { }
```

  そして、Visitor クラスはこのインタフェースを実装した上で、すべての種別に対応する `visit種別()` を定義すればよい。

なんだかごちゃごちゃで頭がこんがらがりますが、このようにすれば個々のノードは `Visitor` インタフェースの変更の影響を受けませんし (そもそも `Visitor` インタフェースはからっぽで変更されません)、Visitor オブジェクトも自分が取り扱う種類のインタフェース群だけ対処すればいいわけです (といっても `AllVisitor` を実装することが結局多そうな気がしますが…)。

4 SableCCコンパイラフレームワーク

4.1 SableCCとは…

SableCCは、米国マギル大学の院生(当時)Étienne Gagnonが1997年に修論で作ったコンパイラコンパイラです。修論作品ではありますが、よくできていて使いやすいので一定のファンがいて、今でもメンテナンスされ続けています。どういうところが「よくできている」かというと…

そもそもコンパイラコンパイラとは「コンパイラを作ることをある程度まで自動化するツール」ですが、全部が自動化できるわけはありませんね。自動化できる部分はおおむね、字句解析部と構文解析部で、これらのコードを自動生成するツールが昔から作られて来ました。Unix標準のツールであるLex、Yaccがこのようなものの起源だと言えます。

そして問題は、自動生成される部分と「自分が作る部分」とのインタフェースがしやすいかどうかです。たとえばYaccではC言語のコードと文法を「混ぜて」書くので、その部分でいまひとつ書きやすくないところがあります。

これに対しSableCCでは、構文木の生成までを自動でおこない、そのあとの「たどる」部分はユーザ(コンパイラ作成者)が書くという分担です。なので、受渡しはできあがった木という点できれいに分離されていて分かりやすく、また木の処理についても上述の改良されたVisitorパターンを活用するなどにより、あまり大変でなく構文木の処理を行えるので使いやすいわけです。

とりあえずごたくはそれくらいにして、以下では「字句解析」「構文解析」「構文木のたどり」の3段階に分けてSableCCの使い方と機能を見て行きます。

4.2 字句解析

字句解析部とは、入力ファイル(コンパイラの場合はソースコード)を読み込みながら、「名前」「数値」「記号」などの「かたまり」に分解して行く処理を言います。ここではまず文法は最小限にして、名前と数値リテラルを取得できるような字句定義を書いてみます。

```
Package tok1;           ← Javaパッケージ名

Helpers                 ←以下補助定義
  digit = ['0'..'9'] ; ←文字集合(数字)
  lcase = ['a'..'z'] ; ←文字集合(小文字)
  ucase = ['A'..'Z'] ; ←文字集合(大文字)
  letter = lcase | ucase ;  ←英字

Tokens                             ←以下端記号の定義
  ident = letter (letter|digit)* ; ←名前の定義
  number = ('+'|'-'|) digit+ ;     ←定数の定義
  blank = (' '|13|10)+ ;           ←空白

Ignored Tokens                     ←空白は無視する
  blank;

Productions                        ←以下文法
  prog = {empty}                   ←{...}については後で説明
       | {ident} prog ident
       | {number} prog number
       ;
```

これがたとえば`tok1.grammer`というファイルに入っていたとすると、「`sablecc tok1.grammer`」でこのファイルを処理し、必要なクラス群を生成します。エラーと言われなければ OK です。

コンパイラドライバはここでは`Compiler`というクラス名で作成してみました。単にファイルを用意し、構文解析を呼び出し、`Executor`という名前で作った Visitor を用意して`apply()`するだけです (先の説明での`accept()`に相当)。

```
package tok1;
import tok1.parser.*;
import tok1.lexer.*;
import tok1.node.*;
import java.io.*;
import java.util.*;

public class Compiler {
  public static void main(String[] args) throws Exception {
    Parser p = new Parser(new Lexer(new PushbackReader(
      new InputStreamReader(new FileInputStream(args[0]), "JISAutoDetect"),
        1024)));
    Start tree = p.parse();
    Executor exec = new Executor();
    tree.apply(exec);
  }
}
```

`Executor`は次の通り、単に名前や数値が visit されたときにその対応する文字列を表示するだけです。

```
package tok1;
import tok1.analysis.*;
import tok1.node.*;

class Executor extends DepthFirstAdapter {
  public void outAIdentProg(AIdentProg node) {
    System.out.println("ident: " + node.getIdent().getText());
  }
  public void outANumberProg(ANumberProg node) {
    System.out.println("number: " + node.getNumber().getText());
  }
}
```

簡単なプログラム (?) で試してみます。

```
% sablecc tok1.grammer
...
% javac tok1/Compiler.java
(古い構文が一部にあるので警告メッセージが出るかも)
% cat tok1.txt
% cat tok1.txt
aaaa bbbb 1111
2345
% java tok1.Compiler tok1.txt
```

```
ident: aaaa
ident: bbbb
number: 1111
number: 2345
%
```

演習 4 この例をそのまま動かしなさい。動いたら、名前や数値の定義を改良してみなさい。また、次の語句も追加してみてください。それぞれの定義は自分で考えてみてください。

a. 実数定数を追加する。
b. 文字列定数を追加する。
c. コメントを追加する。

4.3 構文解析

次は構文解析ですが、要するに BNF で記述された構文に対して、入力 (を字句解析でかたまりに分けたもの) をあてはめて構文木を作る作業が構文解析ということになります。今度は構文をもうちょっとそれらしく書いた例を示します。

```
Package syn1;

Helpers
  digit = ['0'..'9'] ;
  lcase = ['a'..'z'] ;
  ucase = ['A'..'Z'] ;
  letter = lcase | ucase ;

Tokens
  number = ('+'|'-'|) digit+ ;
  blank = (' '|13|10)+ ;
  if = 'if' ;
  read = 'read' ;
  print = 'print' ;
  semi = ';' ;
  assign = '=' ;
  lt = '<' ;
  gt = '>' ;
  lbra = '{' ;
  rbra = '}' ;
  lpar = '(' ;
  rpar = ')' ;
  ident = letter (letter|digit)* ;

Ignored Tokens
  blank;

Productions
  prog = {stlist} stlist
```

```
        ;
    stlist = {empty}
         | {stat} stlist stat
         ;
    stat = {assign} ident assign expr semi
         | {read}   read ident semi
         | {print}  print expr semi
         | {if}     if lpar cond rpar stat
         | {block}  lbra stlist rbra
         ;
    cond = {gt} [left]:expr gt [right]:expr
         | {lt} [left]:expr lt [right]:expr
         ;
    expr = {ident} ident
         | {number} number
         ;
```

なお、条件のところに left とか right とか書かれていますが、これは SableCC では BNF の右辺に同じ記号名が 2 回現れるときはそれを区別する名前を前置する必要があるためです。

コンパイラドライバは先のものと同じですが、実行部分はないのでむしろ簡単です。

```
package syn1;
import syn1.parser.*;
import syn1.lexer.*;
import syn1.node.*;
import java.io.*;
import java.util.*;

public class Compiler {
  public static void main(String[] args) throws Exception {
    Parser p = new Parser(new Lexer(new PushbackReader(
      new InputStreamReader(new FileInputStream(args[0]), "JISAutoDetect"),
        1024)));
    Start tree = p.parse();
  }
}
```

簡単な例で試してみます。

```
% sablecc syn1.grammer
...
% javac syn1/Compiler.java
...
% cat syn1.txt
x = 10;
read y;
print x;
if(x > y) {
 print y;
}
```

```
% java syn1.Compiler syn1.txt
%
```

「何も言わない」ということは構文解析が成功したという意味になります。

演習 5 上の例をそのまま動かしてみよ。動いたら、いく通りかのプログラムを打ち込み、構文エラーは構文エラーとして検出されるこも確認しなさい。その後、次のような変更を行ってみなさい。

a. 加減乗除のできる式の構文定義を追加してみなさい。
b. while 文などの構文を入れてみなさい。
c. if 文に else 部がつけられるようにしてみなさい。

4.4 構文木のたどり

いよいよ、構文木をたどって動作を行う部分を見てみましょう。既に述べてきたように、SableCC では構文木はパーサによって自動的に作られ、それをたどるのには拡張された Visitor パターンが使われています。Visitor の土台となるクラスとして `DepthFirstAdapter` というクラスが生成されていて、ここには文法に現れるすべてのノードの visit メソッドが予め「何もしない」形で用意されているので、このクラスを継承して必要なところだけをオーバライドしていくことで必要な処理を記述します。

オーバライドするためには、メソッド名が分かっている必要がありますね。SableCC では、構文規則に対応してメソッド名が次のように決められます。まず構文規則が次のものだとします。

```
xxx : {yyy} aa bb cc
    | {zzz} [left]:aa bb [right]:aa
    ;
```

まずノードクラスについて説明しましょう。1 つのノードは 1 つの規則に対応しているので、上の場合は 2 つのノードオブジェクトが定義されています。それらのクラス名はそれぞれ、「`AYyyXxx`」と「`AZzzXxx`」になります (先頭が A、次が {} 内に書かれた規則の名前を Capitalize したもの、次が左辺の記号名を Capitalize したもの)。

そして、これらのノードクラスはそれぞれ、右辺の各要素を取り出すメソッドとして `getAa()`、`getBb()`、`getCc()` の 3 つ、および `getLeft()`、`getBb()`、`getRight()` の 3 つを持ちます (このため、同じ名前の記号に対しては区別のための別の名前を指定する必要があったわけです)。これらが返すのはそれぞれのノードオブジェクトですが、そのノードが端記号の場合はその端記号に対応していた文字列が `getText()` によって取得できます。

いよいよ Visitor のためのメソッドですが、これは `DepthFirstAdapter` において各ノードごとに 3 つのメソッドが用意されています。たとえば上の例で 1 番目のノードでは次のようになります。

```
public void inAYyyXxx(AYyyXxx node) { ... }
public void outAYyyXxx(AYyyXxx node) { ... }
public void caseAYyyXxx(AYyyXxx node) { ... }
```

構文木は名前通り深さ優先順でたどられますが、最初にそのノードに到達するときに `in` メソッドが呼ばれ、最後にそのノードから出ていくときに `out` メソッドが呼ばれ、その間で子ノードに対する `apply()` が呼ばれます。多くのノードはこの「最初」「最後」だけで用が足りるのですが、「途中」でも処理が必要な場合は `case` メソッドをオーバライドして使用します。ただし `case` メソッドをオーバライドした場合、その中で自分で子ノードの `apply()` を呼ばなければ、子ノードはたどられません (したがって、たどりたくない場合にも `case` をオーバライドします)。つまり、次のようにするのが標準です。

```
@Override  ←名前を間違えやすいので必ずこのアノテーションをつける
public void caseAYyyXxx(AYyyXxx node) {
  // 最初に到達したときの処理…
  node.getAa().apply(this);
  // aa と bb の間の処理…
  node.getBb().apply(this);
  // bb と cc の間の処理…
  node.getCc().apply(this);
  // 終って出て行くときの処理
}
```

さて、これでオーバライドのしかたは分かりましたが、あと 1 つ説明すべきことが残っています。構文木をたどりながら処理をするとき、ノード間でデータを受け渡していくのが普通ですが、メソッドの形は上のように決まっているので、受け渡すデータのためのパラメタを追加することができません。この問題に対処するため、SableCC では `DepthFirstAdapter` において、データの受け渡し用に、次のメソッドを用意しています。

```
void setIn(Node node, Object x);
Object getIn(Node node);
void setOut(Node node, Object x);
Object getIn(Node node);
```

ここで In 側は木の上側から葉に向かってデータを流すのに使い、Out 側は葉から上側に向かってデータを戻すのに使うという想定です。格納されるのは `Object` 値なので、適宜キャストが必要です (古い Java のコンテナのスタイル)。

では具体的に見てみましょう。文法記述は次の通り。

```
Package sem1;

Helpers
  digit = ['0'..'9'] ;
  lcase = ['a'..'z'] ;
  ucase = ['A'..'Z'] ;
  letter = lcase | ucase ;

Tokens
  number = ('+'|'-'|) digit+ ;
  blank = (' '|13|10)+ ;
  if = 'if' ;
  while = 'while' ;
  read = 'read' ;
  print = 'print' ;
  semi = ';' ;
  assign = '=' ;
  add = '+' ;
  sub = '-' ;
  lt = '<' ;
  gt = '>' ;
  lbra = '{' ;
  rbra = '}' ;
```

```
  lpar = '(' ;
  rpar = ')' ;
  ident = letter (letter|digit)* ;

Ignored Tokens
  blank;

Productions
  prog = {stlist} stlist
       ;
  stlist = {empty}
       | {stat} stlist stat
       ;
  stat = {assign} ident assign expr semi
       | {read}   read ident semi
       | {print}  print expr semi
       | {if}     if lpar cond rpar stat
       | {while}  while lpar cond rpar stat
       | {block}  lbra stlist rbra
       ;
  cond = {gt} [left]:expr gt [right]:expr
       | {lt} [left]:expr lt [right]:expr
       ;
  expr = {term} term
       | {add}  expr add term
       | {sub}  expr sub term
       ;
  term = {ident} ident
       | {number} number
       ;
```

コンパイラはこれまでと変わりません。

```
package sem1;
import sem1.parser.*;
import sem1.lexer.*;
import sem1.node.*;
import java.io.*;
import java.util.*;

public class Compiler {
  public static void main(String[] args) throws Exception {
    Parser p = new Parser(new Lexer(new PushbackReader(
      new InputStreamReader(new FileInputStream(args[0]), "JISAutoDetect"),
        1024)));
    Start tree = p.parse();
    Executor exec = new Executor();
    tree.apply(exec);
  }
```

```
}
```

そして Executor で構文木をたどりながら実際の解釈実行を行います。

```
package sem1;
import sem1.analysis.*;
import sem1.node.*;
import java.io.*;
import java.util.*;

class Executor extends DepthFirstAdapter {
  Scanner sc = new Scanner(System.in);
  PrintStream pr = System.out;
  HashMap vars = new HashMap();
  @Override
  public void outAIdentTerm(AIdentTerm node) {
    String s = node.getIdent().getText().intern();
    if(!vars.containsKey(s)) vars.put(s, new Integer(0));
    setOut(node, vars.get(s));
  }
  @Override
  public void outANumberTerm(ANumberTerm node) {
    setOut(node, new Integer(node.getNumber().getText()));
  }
  @Override
  public void outATermExpr(ATermExpr node) {
    setOut(node, getOut(node.getTerm()));
  }
  @Override
  public void outAAddExpr(AAddExpr node) {
    setOut(node, new Integer(((Integer)getOut(node.getExpr())).intValue() +
                             ((Integer)getOut(node.getTerm())).intValue()));
  }
  @Override
  public void outASubExpr(ASubExpr node) {
    setOut(node, new Integer(((Integer)getOut(node.getExpr())).intValue() -
                             ((Integer)getOut(node.getTerm())).intValue()));
  }
  @Override
  public void outAGtCond(AGtCond node) {
    setOut(node, new Boolean(((Integer)getOut(node.getLeft())).intValue() >
                             ((Integer)getOut(node.getRight())).intValue()));
  }
  @Override
  public void outALtCond(ALtCond node) {
    setOut(node, new Boolean(((Integer)getOut(node.getLeft())).intValue() <
                             ((Integer)getOut(node.getRight())).intValue()));
  }
  @Override
```

```
  public void outAAssignStat(AAssignStat node) {
    String s = node.getIdent().getText().intern();
    vars.put(s, getOut(node.getExpr()));
  }
  @Override
  public void outAReadStat(AReadStat node) {
    String s = node.getIdent().getText().intern();
    pr.print(s + "> ");
    vars.put(s, sc.nextInt()); sc.nextLine();
  }
  @Override
  public void outAPrintStat(APrintStat node) {
    pr.println(getOut(node.getExpr()).toString());
  }
  @Override
  public void caseAIfStat(AIfStat node) {
    node.getCond().apply(this);
    if(((Boolean)getOut(node.getCond())).booleanValue()) {
      node.getStat().apply(this);
    }
  }
  @Override
  public void caseAWhileStat(AWhileStat node) {
    while(true) {
      node.getCond().apply(this);
      if(!((Boolean)getOut(node.getCond())).booleanValue()) return;
      node.getStat().apply(this);
    }
  }
}
```

基本的に、式の中では、それぞれの式の値を out メソッドで計算して setOut() でノードの出力値として保持します。中間のノードは子ノードの値を取って来て必要に応じて計算し、自ノードの値とします。read 文や print 文はその場でそれぞれの動作をします。if 文や while 文は、条件部をまず実行し、その結果に応じて本体部の実行を制御するわけです。

実行例は次の通り (フィボナッチ数を指定最大値まで表示します)。

```
% cat sem1.txt
read max;
x0 = 0;
x1 = 1;
while(x1 < max) {
  x2 = x0 + x1;
  x0 = x1;
  x1 = x2;
  print x0;
}
% java sem1.Compiler sem1.txt
max> 10
```

```
1
1
2
3
5
8
%
```

演習 6 上の例をそのまま動かしなさい。動いたら、いく通りかのプログラムを打ち込み、思った通りに動作することを確認しなさい。その後、言語に次のような変更を行ってみなさい。

a. 乗除算も追加してみなさい。
b. do-while 文のような「下端で条件を調べるループ」を追加してみなさい。
c. if 文に else 部がつけられるようにしてみなさい。

演習 7 この方式で自分の好きな言語を設計して実装してみなさい。