システムソフトウェア特論'17 # 9 コンパイラコンパイラ

久野　靖 *

2017.9.15

1　コンパイラコンパイラとは

コンパイラコンパイラ (compiler compiler) とは「コンパイラを生成する翻訳系」という意味です。しかし実際には、何かちょっと記述したらコンパイラが完全にできあがる、というのは現状では困難です。

これまで見て来たように、フロントエンド (字句解析、構文解析、そして意味解析の入口程度) であれば、コンパクトな記述から処理系を生成できるので、コンパイラコンパイラという用語はその範囲をおこなうものとして長く使われて来ました (図 1 左)。類似した用語として**パーサジェネレータ** (parser generator) がありますが、生成するのがパーサだけではないツールも多いので少し意味が違うかも知れません。

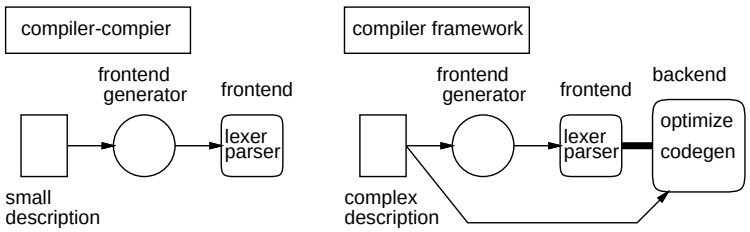

図 1: コンパイラコンパイラの概念

近年では、最適化の技術が進歩し、ある程度の記述 (実際にはかなり専門性が必要) を行なえば最適化を含んだバックエンドまで実現できるようなツールも現れています (LLVM などが有名)。この場合、コンパイラ作成者は共通の中間コードと (まだ無ければ) ターゲットマシン記述を生成し、ツールに含まれるバックエンドがそれを処理するので、コンパイラが生成されるというよりは、コンパイラを構築するツール群、という方が合っています。このようなものは、**コンパイラフレームワーク** (compiler framework) と呼ばれることもあります (図 1 右)。また、フロントエンドはどうやって作ってもいいということから、バックエンドしか関与しないコンパイラフレームワークもあります (LLVM はこれです)。

今回はフロントエンドを生成するという意味でのコンパイラコンパイラの例として、SableCC を取り上げます。ただしその前に、いくつかの前置き (予習) が必要なので、それから始めましょう。

2　いくつかの準備

2.1　Java の動的な型の扱い

Java では親クラスの型の変数には子クラスのインスタンスが入れられるということを以前に説明しました。ということは何が起きているかというと、メソッドを呼び出す時に実際に呼び出されるメ

*電気通信大学 情報理工学研究科

ソッドは実行時に変数にどのクラスのインスタンスが入っているかによって異なる、ということになります。たとえば、次の例を見てください。

```
class A {
  ...
  public void method1(...) { .... (A) .... }
}
class B extends A {
  ...
  public void method1(...) { .... (B) .... }
  public void method2(...) { .... }
}
```

ここで、A 型の変数 x があり、x.method1(...); という呼び出しが書かれていたとします。このとき、実際に動くコードは、x にクラス A のインスタンスが入っていれば (A) であり、クラス B のインスタンスが入っていれば (B) ということになります。また A の子クラスがほかにもあれば、そこで定義されているメソッドのコードかも知れません。このように、実行時にならないと対象が決まらないようなものを**動的束縛** (dynamic binding) と呼びます。

次に、クラス B に含まれているメソッド method2() について考えて見ましょう。これは、B 型の変数に入っている B のインスタンスに対してはもちろん普通に呼べます。しかし、クラス A にはこのメソッドは定義されていないので、A 型の変数に入れてしまうと呼ぶことができません (コンパイルエラーになる)。

```
A x = new B(...);    // B のインスタンスは A 型に代入 OK
x.method2(...);      // エラー: A 型には method2 がない
```

ではどうしたらいいかというと、x に入っているオブジェクトを B 型の変数に入れ直せばいいのです。ただし、このときは「親クラスの型から子クラスの型に」変換するので、そのままではだめで、キャストを書く必要があります。

```
B y = (B)x;         // x の中の値を B 型にキャスト
y.method2(...);     // OK
```

このキャストを**ダウンキャスト** (downcasting) と呼び、このときにインスタンスが実際にクラス B(またはその子クラス) のインスタンスであることをチェックします (それ以外であれば例外 ClassCastException が発生)。このように、Java ではオブジェクトに付随するクラスの情報を用いて実行時にチェックが行なわれます。

ところで、Java ではすべてのオブジェクトはクラス Object を継承している、という話は前にしました。ということは、すべてのクラスは Object のサブクラスであり、従って Object 型の変数に入れられます。これはコンテナなど「何でも入れられる」データ構造を作るのには便利なのですが、取り出して来た後「元の型に戻す」のにはやはりダウンキャストが必要です。

2.2 インタフェースとその扱い

Java の特徴的な機能として、インタフェースがあります。インタフェースとは「実装のない、メソッドの名前と型だけを規定したもの」だと言えます。たとえば次のコードを見てください。

```
interface Doable {
  public void doit(...);
}
```

```
class X implements Doable {
  ...
  public void doit(...) { .... (X) ... }
}
```

この場合、Doable 型の変数 d にクラス X のインスタンスを入れることができます (キャストなしに)。そして、d.doit(...):が呼ばれたときには、(X) のコードが動くわけです。

インタフェースの何がいいのでしょう? 継承と同じようなものでは? いいえ、継承は変数やメソッドを引き継いで来るので、必然的に似たような構造のクラス (あるクラスを拡張したクラス) を作ることにしか使えません。まったく無関係な 2 つのクラスでは駄目なのです。

しかしインタフェースは「このようなメソッドを持つ」ということだけが条件なので、まったく別のクラスに同じインタフェースを実装 (implements) させることができますし、1 つのクラスが多数のインタフェースを実装しても問題ないです (継承は 1 つの親クラスに限定)。

それでは、X クラスのインスタンスを Object 型の変数 z に入れてあるとき、これを Doable 型の変数 d に入れるには? それもやはりダウンキャストを使います (クラス階層の上下とはもはや無関係ですが)。これもやはり、実行時のクラス情報でチェックされます。

```
Doable d = (Doable)z;  // ダウンキャスト:実行時にチェック
```

2.3 Visitor パターンとその拡張

ここまでに使って来た抽象構文木の木構造を見て「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。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 eval() として書かれています (図 2)。

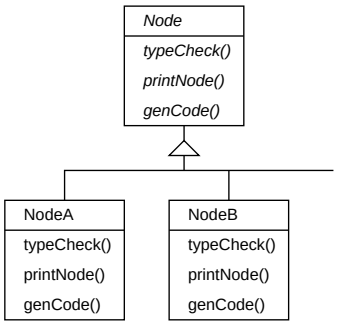

図 2: 素朴な再帰による木構造の処理

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 3)。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。

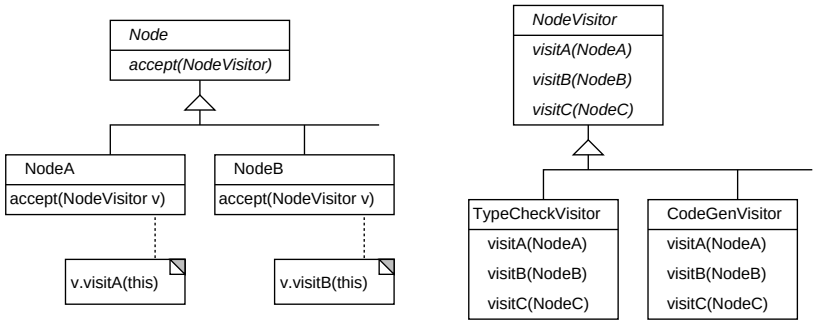

図 3: Visitor パターンへの変換

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いう指摘があります。

> 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…)。

3 SableCC コンパイラコンパイラ

3.1 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。元が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…

ここまでに字句解析器 JFlex とパーサジェネレータ cup を見て来ました。これらはうまく連携できますが、やはり別々のソフトなので 2 つのファイルを取り扱う必要があります。あと、パーサ側から「何行目の何文字目にエラー」みたいなメッセージを出そうとすると、文字を読むのは字句解析器の担当なので、連携が複雑になりがちです。

また、cup では還元が起きるときの動作を構文記述と一緒に書いていましたが、ということはその動作が起きるタイミングは固定で 1 回だけということになります。ですから通常は、木構造を組み立ててそこで処理をするということになります。また、文法記述と動作が混ざって書かれているので読みづらいという弱点があります。

これに対し、SableCC は次のようにできています。

- 構文解析と字句解析が一体で 1 つの記述ファイルから生成し、連携も内部で自動的になされる。

- 構文記述にアクションを書く必要はなく、構文木は自動で生成される。
- 生成された構文木は前述の Visitor パターンに合った形になっていて、コンパイラ作成者はさまざまな Visitor クラスを作ることで繰り返しソース情報を処理できる (マルチパス)。

以下ではまず記述ファイルの書き方、次に構文木のたどりの 2 段階に分けて SableCC の使い方と機能を見て行きます。

3.2 記述ファイルの書き方

SableCC では生成ソースを 1 つのパッケージに入れるので、たとえばパッケージが `sam91` であれば、すべてのソースはディレクトリ `sam91/`以下に置かれます。自分が書くソースもそうすることになります。作成するファイルは冒頭に「`package sam91;`」を入れ、置き場所は「`sam91/`なんとか`.java`」で、そのファイル名でコンパイルします。そして実行開始するときは「`java sam91.` なんとか」になります (クラス名を指定するので)。

SableCC の記述ファイルは上述のように字句解析と構文解析を一緒に書くので、いくつかのセクションに分かれています。簡単な具体例で見てみましょう。

```
Package sam91;

Helpers
  iconst = ['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 = {stat} stlist stat
```

```
        | {empty}
        ;
  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
       | {iconst} iconst
       ;
```

規則のあちこちに [名前]: というものが書かれていますが、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つける必要があるためです。

main() は次のように、PushbackReader クラスを渡すようになっています。ここでは解析するだけなので、構文チェッカができます。

```
package sam91;
import sam91.parser.*;
import sam91.lexer.*;
import sam91.node.*;
import java.io.*;
import java.util.*;

public class Sam91 {
  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();
  }
}
```

では実行を前回の例で試してみます。

```
% sablecc sama1.grammer
...
% javac sam91/Sam91.java
...
% cat test.min
read x; read y;
if(x > y) { z = x; x = y; y = z; }
print x; print y;
% java sam91.Sam91 test.min
%
```

「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 9-1 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. while 文を追加してみなさい。
b. 算術式の計算ができるようにしてみなさい。
c. if 文に else 部がつけられるようにしてみなさい。
d. その他好きな拡張をしてみなさい。

なお、SableCC には cup のような「あいまい文法」の機能がないため、文法を曖昧さの無い BNF で書く必要があり、記述がやや面倒です。

3.3 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
}
```

なお、上の case... をオーバライドしなかった場合は、子ノードの処理を呼ぶ前と後にそれぞれ次のメソッドを呼ぶという定義が有効です。

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

これらはそれぞれ、木をたどって来てノードに入って来た時と、ノードから出て行く時に呼ばれるメソッドということになります。入口だけ、出口だけで処理が必要なら、こちらをオーバライドするのが簡単で。これもオーバライドしなかったら? そのときはこれらの「何も動作がない」版が動きます。

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 sam92;

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

```
Tokens
  iconst = ('+'|'-'|) 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 = {stat} stlist stat
         | {empty}
         ;
  stat = {assign} ident assign expr semi
       | {read}   read ident semi
       | {print}  print expr semi
       | {if}     if lpar expr rpar stat
       | {while}  while lpar expr rpar stat
       | {block}  lbra stlist rbra
       ;
  expr = {gt}  [left]:nexp gt [right]:nexp
       | {lt}  [left]:nexp lt [right]:nexp
       | {one} nexp
       ;
  nexp = {add}  nexp add term
       | {sub}  nexp sub term
       | {term} term
       ;
  term = {ident}  ident
       | {iconst} iconst
       ;
```

main() では解析木を取得して Executor で実行します。

```
package sam92;
import sam92.analysis.*;
import sam92.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 outAIconstTerm(AIconstTerm node) {
    setOut(node, new Integer(node.getIconst().getText()));
  }
  @Override
  public void outAOneNexp(AOneNexp node) {
    setOut(node, getOut(node.getTerm()));
  }
  @Override
  public void outAAddNexp(AAddNexp node) {
    int v = (Integer)getOut(node.getNexp()) + (Integer)getOut(node.getTerm());
    setOut(node, new Integer(v));
  }
  @Override
  public void outASubNexp(ASubNexp node) {
    int v = (Integer)getOut(node.getNexp()) - (Integer)getOut(node.getTerm());
    setOut(node, new Integer(v));
  }
  @Override
  public void outAOneExpr(AOneExpr node) {
    setOut(node, getOut(node.getNexp()));
  }
  @Override
  public void outAGtExpr(AGtExpr node) {
    if((Integer)getOut(node.getLeft()) > (Integer)getOut(node.getRight())) {
      setOut(node, new Integer(1));
    } else {
      setOut(node, new Integer(0));
    }
```

```
    }
    @Override
    public void outALtExpr(ALtExpr node) {
      if((Integer)getOut(node.getLeft()) < (Integer)getOut(node.getRight())) {
        setOut(node, new Integer(1));
      } else {
        setOut(node, new Integer(0));
      }
    }
    @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.getExpr().apply(this);
      if((Integer)getOut(node.getExpr()) != 0) { node.getStat().apply(this); }
    }
    @Override
    public void caseAWhileStat(AWhileStat node) {
      while(true) {
        node.getExpr().apply(this);
        if((Integer)getOut(node.getExpr()) == 0) { 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 test2.min
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 9-2 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 9-3 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。

3.4 言語の見た目とその効果

上で見たのは比較的「普通の」構文を持つおもちゃ言語でしたが、本質は同じでも、もっと見た目を変えることができます。そこで、日本語を使ったヘンな言語を定義してみます。

```
Package sam93;
Helpers
  digit = ['0'..'9'];
  lcase = ['a'..'z'];
  ucase = ['A'..'Z'];
  letter = lcase | ucase;
Tokens
  woireru = 'を入れる';
  niyomikomu = 'に読み込む';      ←「に」が先だとまずいので注意
  ni = 'に';                        (先に書かれているものが優先なので)
  wouchidasu = 'を打ち出す';
  naraba = 'ならば';
  wojikkou = 'を実行';
  noaida = 'の間';
  gt = '>';
```

```
  lt = '<';
  add = '+';
  sub = '-';
  number = digit+;
  ident = letter (letter|digit)*;
  blank = (' ' | 10 | 13 )+;

Ignored Tokens
  blank;

Productions
  prog = {stlist} stlist
       ;
  stlist = {empty}
       | {stat} stlist stat
       ;
  stat = {assign} ident ni expr woireru
       | {read}   ident niyomikomu
       | {print}  expr wouchidasu
       | {if}     expr naraba stlist wojikkou
       | {while}  expr noaida stlist wojikkou
       ;
  expr = {term} term
       | {gt} [left]:term gt [right]:term
       | {lt} [left]:term lt [right]:term
       ;
  term = {fact} fact
       | {add}  term add fact
       | {sub}  term sub fact
       ;
  fact = {ident} ident
       | {number} number
       ;
```

この言語のプログラム例を示します。トークンの間は空けてもいい (というか空けた方が読みやすい) のですが、日本語ぽくわざとくっつけて書いてみました。

```
n に読み込む
x に 1 を入れる
n>0 の間 x に x+x を入れる n に n-1 を入れるを実行
x を打ち出す
```

演習 9-4 次のような計算をするプログラムをこの言語で書け。

a. 2 つの数を読み込んで合計を打ち出す。
b. 2 つの数を読み込んで大きい順に打ち出す (2 数は等しくないものとしてよい)
c. 入力した値 n を超えないフィボナッチ数を順番に打ち出す。

コンパイラドライバは先と変わらないので、ツリーインタプリタのみを示します。

```
package sam93;
import sam93.analysis.*;
import sam93.node.*;
import java.io.*;
import java.util.*;

class Executor extends DepthFirstAdapter {
  Scanner sc = new Scanner(System.in);
  PrintStream pr = System.out;
  HashMap<String,Integer> vars = new HashMap<String,Integer>();
  @Override
  public void outAAssignStat(AAssignStat node) {
    vars.put(node.getIdent().getText(), (Integer)getOut(node.getExpr()));
  }
  @Override
  public void outAReadStat(AReadStat node) {
    String s = node.getIdent().getText();
    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.getExpr().apply(this);
    if((Integer)getOut(node.getExpr()) != 0) { node.getStlist().apply(this); }
  }
  @Override
  public void caseAWhileStat(AWhileStat node) {
    while(true) {
      node.getExpr().apply(this);
      if((Integer)getOut(node.getExpr()) == 0) { break; }
      node.getStlist().apply(this);
    }
  }
  @Override
  public void outATermExpr(ATermExpr node) {
    setOut(node, getOut(node.getTerm()));
  }
  @Override
  public void outAGtExpr(AGtExpr node) {
    if((Integer)getOut(node.getLeft()) > (Integer)getOut(node.getRight())) {
      setOut(node, new Integer(1));
    } else {
```

```
      setOut(node, new Integer(0));
    }
  }
  @Override
  public void outALtExpr(ALtExpr node) {
    if((Integer)getOut(node.getLeft()) < (Integer)getOut(node.getRight())) {
      setOut(node, new Integer(1));
    } else {
      setOut(node, new Integer(0));
    }
  }
  @Override
  public void outAFactTerm(AFactTerm node) {
    setOut(node, getOut(node.getFact()));
  }
  @Override
  public void outAAddTerm(AAddTerm node) {
    int v = (Integer)getOut(node.getTerm()) + (Integer)getOut(node.getFact());
    setOut(node, new Integer(v));
  }
  @Override
  public void outASubTerm(ASubTerm node) {
    int v = (Integer)getOut(node.getTerm()) - (Integer)getOut(node.getFact());
    setOut(node, new Integer(v));
  }
  @Override
  public void outAIdentFact(AIdentFact node) {
    setOut(node, vars.get(node.getIdent().getText()));
  }
  @Override
  public void outANumberFact(ANumberFact node) {
    setOut(node, Integer.parseInt(node.getNumber().getText()));
  }
}
```

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。

では、さっきのプログラムを実行してみます。

```
% cat test.min
n に読み込む
x に 1 を入れる
n>0 の間 x に x+x を入れる n に n-1 を入れるを実行
x を打ち出す
% java sam93.Sam93 test.min
```

```
n> 8
256
%
```

やっていることは前の言語と同じですが、見た目が違うとそれだけで、それを読んだり書いたりする人にとっての「違い」もはっきり現れます。皆様はこれを見てどう思いましたか?

演習 9-5 上の例をそのまま動かしなさい。動いたら、演習 9-4 で書いたプログラムを動かしてみなさい。その後、言語に次のような変更を行ってみなさい。

a. if 文に else 部が書けるように直して、その上で「2 数の最大」を動かしてみなさい。
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. その他、好きな変更を行ってみなさい。

4 課題 9A

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# 9」、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. SableCC のようなコンパイラコンパイラについて、使ってみてどのように思いましたか。
Q2. Visitor パターンについて知っていましたか。使っ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。