プログラミング言語各論
第2回レポート

2005 年度担当: 中井央

2005 年 10 月 20 日

1 レポート課題

以下の手順にしたがって順次プログラムを作っていってもらう。最終的にでき上がるのは、コマンドラインから与えられた算術式 (中置記法) を受けとり、内部でその式の木構造を作り、Visitor パターンによってその木を渡り歩きながら処理をするものである。渡り歩きながら行なう処理は、与えられた中置記法を前置記法に変換すると後置記法へ変換するものである。

1.1 準備

今回のレポートにあたって、レポート用のディレクトリを次のように構成せよ。

1. レポート全体用のディレクトリを report2 とする。
2. Visitor パターンの課題用のディレクトリ Visitor を report2 の中に作る。
3. 字句解析の課題用のディレクトリ Lexer を report2 の中に作る。
4. 構文解析の課題用のディレクトリ Parser を report2 の中に作る。
5. コマンドライン引数を解析し、その結果を Visitor パターンを用いて走査する課題用のディレクトリを Last とする。

1.2 Visitor パターンの課題

通常、我々が演算式を記述する時は、中置記法を用いる。つまり、1+2*3 のように被演算子の間 (中) に演算子を置く記述方法を用いている。ところで、まず演算子を置き、その後ろに被演算子を置く前置記法、被演算子の並びの後、演算子を置く後置記法も存在する。1+2*3 に対応する前置記法、後置記法はそれぞれ、+1*23、123*+ である。1+2*3 の木を書いてみると分かるが、前置記法では、その木のルート (+) からみて、左の子が 1 で、右の子が*、*からみてその左の子が 2、右の子が 3 である、と読める。

後置記法は 1 に 2 と 3 をかけたもの (*) を足す (+) と読める。

木の訪問には、ルートノードから始めて、その子供を左から順に訪問するが、それぞれの子にさらに子 (孫) がある場合はその子 (孫) を先に訪問し、帰ってきてから自身の弟 (妹) を訪問する、という方法がある。これを深さ優先の訪問という。

式を木構造で表現している場合、深さ優先で各ノードを訪問する際、ノードを訪問した時 (行きがけ) に挙動を起こすか、ノードから去って親ノードへ戻る時 (帰りがけ) に挙動を起こすかで結果が異なる。

前置記法を出力するには行きがけにそのノードの記号を出力すれば良い。末端 (数字) の場合は単にその数字を出力するだけで良い。

後置記法は逆に帰りがけに記号を出力すれば良い。

さて、前置きが長くなったが、第 5 回のテキストの演習 1 で扱った式を木構造で表し、演算を行う Composite パターンの適用例をもとにし、それを第 5 回のテキストにあるサンプルを参考にして、Visitor パターンに作り変え、ListVisitor に相当する実際の Visitor クラスを 2 つ作れ。1 つは前置記法を出力するための PreVisitor クラス、もう 1 つは後置記法を出力するための PostVisitor クラスである。

なお、式は演算子として+, -, *, / を扱えることとする。演算子の優先順位は通常の四則演算と同じとする。

これらのクラスを使うための Main クラスを図 1 に示す。

また、main メソッドを書き換えた Main2.java も作成せよ。main メソッドは Main クラスのものとは異なる式を表すようにすること。ただし、演算子を最低 5 つ含むようにせよ。

一連のファイルは上述の Visitor ディレクトリの中に作成せよ。ただし、ここでは node というパッケージにすることにする。すなわち、実際にはさらに node というディレクトリを作り、その中に各ファイルを納めるようにせよ。また、Main クラスはパッケージの外に置いて node パッケージが利用できるかどうかも確認せよ。パッケージを作成するに当たっては次の点に注意すること。

1. パッケージの外側から利用されることを考慮し、各クラス、そのコンストラクタおよびメソッドは public で宣言する。(パッケージ外からアクセスがないものは public にしない)
2. 上記のため、各クラスごとに**classname.java** というファイル名で保存する必要がある。classname の部分には、そのクラスの名前が入る。

```
 1 class Main {
 2   public static void main(String args[]){
 3     Node tmp1, tmp2, tmp3, tmp4, tmp5;
 4
 5     tmp1 = new Digit(1);
 6     tmp2 = new Digit(2);
 7     tmp3 = new Digit(3);
 8
 9     tmp4 = new Mult(tmp2, tmp3);
10     tmp5 = new Plus(tmp1, tmp4);
11
12     System.out.println(``1+2*3 => ``);
13
14     System.out.print(``prefix : ``);
15     tmp5.accept(new PreVisitor());
16     System.out.println(```');
17
18     System.out.print(``postfix : ``);
19     tmp5.accept(new PostVisitor());
20     System.out.println(```');
21   }
22 }
```

図 1: `Main.java` (レポート課題用の Main クラス)

1.3 字句解析プログラムの作成

与えられた式の字句解析を行なうプログラムを作成する。この課題に関するファイルは上述の Lexer ディレクトリの中で作ること。

字句解析とは入力された文字列から意味のある塊 (単語) を切り出す作業である。ここでは入力は記号と記号の間が半角空白で区切られているとする。すなわち、`(1+2)*3` のような入力は `( 1 + 2 ) * 3` のように空白で区切って入力しなければならない。

ここでは字句の切り出しに StreamTokenizer というクラスを利用する。StreamTokenizer は与えられた入力ストリームから、字句の切り出しを行なう。空白、数字、単語というカテゴリを設定できる。ここでは StreamTokenizer を使った字句の切り出しプログラム Lexer.java とそれ用のテストプログラム LexerTest.java を作成する。作成に当たっては Java API のマニュアルを参照すること。

Lexer.java は次のような構成になる。

1. 切り出した字句に関して、それが数ならば値を保持する int 型の val という変数を用意する。
2. 切り出した字句に関して、それが単語ならば値を保持する String 型の s という変数を用意する。
3. StreamTokenizer オブジェクトを保持する変数 st を用意する。
4. 文字列を引数とするコンストラクタを用意する。コンストラクタの構成は次のようになる。
   (a) コンストラクタでは、まず、StreamTokenizer を文字列を対象に解析を行なえるようにインスタンスを作る。このインスタンスを変数 st に入れる。
   (b) 全ての文字 (0 から 65535 まで) を一度、「通常」文字にする。
   (c) 数値を構文解析するように指定する。
   (d) 半角空白を区切り文字として指定する。
   (e) 式に現れる記号 `+`, `-`, `*`, `/`, `(`, `)` を単語を構成する文字 (ワード文字) として指定する。なお、指定に当たっては指定用のメソッドは文字の範囲を受けとるが、その文字からその文字までという指定をして各文字ごとに登録する。
5. 字句解析用メソッド nexttoken を以下のように定義する。これは引数無しで int 型の値を返す。
   (a) メソッドにローカルな int 型の変数 n を宣言す

る。StreamTokenizer オブジェクトから「次のトークン (字句)」を切り出し、一時的に n に入れる。
(b) 切り出した文字列が数ならばその値を val に保持する。
(c) また、単語であるかも知れないのでその文字列を StreamTokenizer オブジェクトから受けとり、s が持つようにする。

※注： val, s は場合わけしなくても常にそれらの値が代入されるようにしておいても良い。

次に LexerTest クラスを作る。構成は次のような感じである。

1. コマンドラインの第一引数を解析の対象とする。なお、コマンドラインから式を与える時は、式全体をダブルクォートで括ること。例えば次のようになる。

   ```
   uni% java LexerTest "1 + 2 * 3"
   ```

2. Lexer オブジェクトを生成する。
3. 字句を切り出し、それが入力の終りでない限り、繰り返し切り出しをするようにする。
4. 各切り出した字句については、その種類を画面に表示する。すなわち、数字を切り出した時には `number = 1` のように表示する。1 の部分には切り出した数の値を入れる。単語を切り出した時にはその文字列を表示する。その他の場合は `other!` と表示する。

完成したら適当に文字列を与えてうまく動くことを確認すること。

1.4 構文解析プログラムの作成

与えられた式を字句に切り分けたら次は、構文として正しいかどうかをチェックする構文解析を行なう。構文解析プログラムは字句解析プログラムと連係して実行される。ここで行なう構文解析では現在の状態に対して次の記号が受け入れられるかどうかが決定できるものである。詳細はコンパイラの教科書などで学ことができるが、ここでは簡単に原理を紹介し、今回の目的のための簡単な構文解析プログラムの作成を行なう。

プログラムの作成は上述の Parser ディレクトリで行なうこと。

まずは構文解析を行なうためのクラスを作成しよう。ここでは四則演算の構文解析を対象としている。これは文法で表すと次のようになる。

```
E : T { + T } ;
T : F { * F } ;
F : (E) | NUM ;
```

この例では 1 行に 1 つの「規則」を書いている。1 つの規則は : で区切られた左辺と右辺にわかれる。1 つの規則の終りを ; で表す。: の左側に現れる記号のことを非終端記号と呼ぶ。: の右側には非終端記号および終端記号の並びが来る。終端記号は非終端記号ではないものであり、実際の入力に現れる「字句」を表している。例えば数を表す文字列は無限に存在するがその種類が数であることがわかればよく、上記ではそれを `NUM` と表現している。

`E` は expression (式) の頭文字である。`T` は term (項) の頭文字である。`F` は fact (要素) の頭文字である。`{` と `}` で囲まれた間にある要素はその並びが 0 個以上あることを意味する。

そうすると第 1 行目は式とは何かを定義していて、式とは、1 つの項の後ろに「 + 記号に続く項」が 0 個以上並んだもの、と言うことを示している。第 2 行目も同様である。第 3 行目は `F` は括弧に囲まれた式か単独の数かということを示している。`|` は選択を意味する。`E` と `T` と `F` を設けているのは演算子に優先順位をつけるためである。例えば入力が `1+2` だったとする。そうすると足し算の演算記号が入っているので形としては一番目の規則 (の右辺) に合う。このとき、右辺の最左にある `T` は結局は `1` に一致するのだが、規則にしたがって、今度は二番目の規則を用いて展開することになる。二番目の規則はかけ算を表現しているのであるが、入力の `1` に一致したいのでここでは `{ * F }` の部分は 0 個であると考える。そうすると二番目の規則の右辺の最左の記号 `F` を更に展開することを考える。それには三番目の規則を使う。三番目の規則は `(E)` (括弧で囲われた式) か単独の数かを表現しているが、`1` に一致するのは `NUM` である。ここまでの処理の過程は次のような感じである。

```
E
|
```

```
T （このあと { + T } で展開する）
|
F
|
1 + 2
```

さて、入力のうち、1 の処理が終った。次は + の処理となる。ここまでの処理を振り替えると、+ が現れるのは一番目の規則を使った時だった。そしてそのうちの最左の T を展開していったがいまはそれが終った段階であり、入力の + と一番目の規則にある + が一致する。そして次は最後の入力 2 と { + T } に現れる T の一致を試みるがこれは 1 に一致させる時と同じ手順を踏めば良い。したがって展開の過程は次のようになる。

```
E
|
T + T
|   |
F   F
|   |
1 + 2
```

さて、1+2*3 の場合、我々の概念では 1+(2*3) のように考える。すなわち、先に 2*3 の計算が行なわれる。図にすると次のような感じである。

```
E
|
T + T
|   |
F   F * F
|   |   |
1 + 2 * 3
```

入力 (最下段) に近い側で結び付いている (この例では F * F の) 部分の方が、その上で結び付いている (この例では T + T) 部分よりも結び付きが強い。

さて、ここまでにやってきたことは式を表す文法規則を見ながら与えられた式に対して、その式になるように文法規則を繰り返し適用したということである。途中でどうやっても適用できなくなったら入力に誤りがあったと言うことになる。

以上で大雑把に構文解析の説明をした。次は構文解析用のクラスの設計をする。次の方針とする。

1. クラス名は Parser とする。
2. 字句解析には先ほど作成した字句解析クラスを使う。ただし、字句解析クラスは今回の目的に合うように変更する必要がある。
3. 構文解析のアルゴリズム自体はプログラミング言語処理系という別の授業のテキスト
   (http://www.slis.tsukuba.ac.jp/~nakai/2005/Compiler/ulisonly/no4.pdf)
   の最後のページのもの (図 4) を流用する。ただし、今回の目的に合うよう、かつ Java で使用するように少し修正が必要である。

1.4.1 Parser クラスの設計・実装

Parser クラスの実装は次のように行なう。

1. Lexer から得られる次の字句を示す変数 nexttoken を用意する。
2. 字句解析器では +, -, *, /, (,) と数字を切り出す。Parser クラスのインスタンスは Lexer クラスのインスタンスに対し、nexttoken() の呼び出しをして次の字句を得る。このとき、切り出した内容に応じてそれぞれ、256, 257, 258, 259, 260, 261, 262 を返すように Lexer を改造する (後述)。これらの数値は生では使いたくないので、Parser クラスで定数として宣言しておく。それぞれの定数名を PLUS, MINUS, MULT, DIV, LPAR, RPAR, NUM とする。
3. 趣味の問題ではあるが…。図 4 の E, T, F をそれぞれ exp, term, fact に置き換える。そして、それぞれのメソッドとして Parser クラスに実装する。(ほとんどコピペ)
4. 今回は四則演算に対応するようにしたい。図 4 では引き算、割算の処理が入っていないが、while 文や if 文の条件にそれらを追加する。
5. fact(F) の switch 文ではエラー処理をしていないので、switch 文の最後に default:を追加し、エラー処理を入れる。ここでは単にエラーメッセージを出力し、終了する。
6. while 文中の if 文は冗長であるので省いても良い。
7. 字句解析からの字句の取得は図 4 では yylex からであるが、これを与えられた Lexer オブジェクトからに変更する。

8. コンストラクタを作る。コンストラクタは Lexer オブジェクトを受けとり、自身が持つ Lexer の変数へ代入する。
9. 最後にこのクラスを利用する人が構文解析を始めるための parse メソッドを作る。このメソッドでは一番最初の字句解析器の呼び出しを行なって、構文解析を開始 (exp の呼びだし) する。なお、exp の呼びだし終了後、入力の終りに達していなければエラーである。

1.4.2 字句解析クラスの変更

字句解析クラスは Lexer ディレクトリからコピーして利用することにする。上述の Parser クラスに合わせるため、これまでに作った Lexer クラスを変更する。なお、差分プログラミングと言う考えがあるが、それはある程度クラスをまとまってきちんと整備されている場合である。現在のように開発途上段階ではクラスを書き直すということも良くある。そして、書き直しではなく継承を利用できるようになっていけばよい。なお、クラス Lexer を継承し、nexttoken を上書きすることで変更することも可能である。

さて。

次のように変更をしよう。

1. 受けとりたいのは数と記号だけである。それ以外が入力にあった場合、入力誤りであるとして排除するようにする。それには TT_NUMBER か TT_WORD かというようなチェックをする。なお、ここには入力の終りを示す TT_EOF も含めておかないといつまでも終了しない可能性がある。
2. 入力から数を切り出した場合、Parser.NUMBER を返すようにする。
3. 入力が WORD だった場合、どの記号かをさらにチェックして、それぞれの記号に対応した Parser クラスで定義している定数を返す。意図しない記号の場合のエラー処理も忘れずに。

1.4.3 テストクラスの作成

ここまでに作った構文解析クラスが正しく動くかテストクラスを作成すること。クラス名を ParserTest とする。コマンドラインから式や間違った式を入力してみて、解析が正しく行なわれることを確認しよう。

1.5 最後の課題

最後に上述の構文解析プログラムと Visitor パターンのプログラムを連係させて、コマンドライン引数を受けとり、それを構文解析し、そのときに与えられた式の木構造を作成し、その木構造に対して Visitor パターンを適用するようにせよ。

作成の方針と注意点を以下に記す。

1. Last ディレクトリに関連ファイルを設置すること。とりあえず、Parser の中身と node ディレクトリをコピーすると良い。
2. Parser.java に手を加えて、構文解析をしている間に木を構築するようにせよ。主に次のようになる。
   (a) node パッケージが使えるように import 文を入れる。
   (b) 構文解析用のメソッドは Node クラスのオブジェクトを返すように変更する。
   (c) 各メソッドでは、その中でのメソッド呼び出しの結果を用いてノードを作成し、呼び出し元に返す。
   (d) テスト用プログラム ParserTest.java も以上に合わせて変更する。すなわち、構文解析によって得られた Node オブジェクトに対して、Visitor パターンを適用するようにする。

2 提出方法など

作成したプログラムは次のいずれかの方法で中井が参照できるようにすること。

1. 作成したファイルは uni 上に設置し、作成したプログラムがあるディレクトリと作成したソースファイルに対し、ユーザ nakai に対し、読めるように設定せよ。他のユーザには読めないように設定せよ。プログラムが完成したら、そのファイルの存在場所 (パス名) をメールにて nakai@slis.tsukuba.ac.jp まで報告せよ。
2. 上記が面倒な (or できない) 人は作成したファイルをメールに添付して送信せよ。この場合、一通のメールに全てのプログラムを添付すること。あ

て先は nakai@slis.tsukuba.ac.jp である。tar でまとめて gzip で圧縮してから送ること。tar の使い方は前回のレポート課題を参照すること。

いずれの場合もメールのサブジェクトは次のように記述すること。uid には自分のユーザ ID を入れること。書式間違えないでね!!

```
[PL2005 uid] 2nd report
```

メール本文には、学籍番号、氏名、授業名とひとこと[1]を記載すること。

また、次の形式で紙媒体のレポートを作成し、学務に用意するレポート提出箱に提出すること。

1. この授業のタイトルおよび「第 2 回レポート」
2. 学籍番号および括弧書きの uni のユーザ名。そして氏名
3. 各プログラムごとにプログラム作成に当たっての次の項目を書くこと。(プログラムリストの解説ではない)
   (a) 苦労した点
   (b) どのようにして問題解決をしたか
   (c) 何を参考にしたか
   (図情の学生なんだから書誌情報はきちんと書くこと!)
   (d) そのプログラムが正しく動くことをどのように確かめたか (根拠と結果)
4. 感想、コメントなど (必須)。

〆切は 11/24 (木) 17:00 とする。提出場所は学務課に設けられたレポート提出箱である。多少遅れても提出すれば減点はするが成績をつける対象にはする。レポートの提出がない場合は成績は出せない。

注意

プログラムを正しく提出できていない場合、こちらからメールにて問い合わることがある。一定期間内に回答がない場合は未提出扱いになるので注意すること。

[1] 普通は「レポートを提出します」くらいは書くもんだ…。携帯のメールと通常のメールでは書き方が違うということくらいは理解して欲しい。