# コンパイラ演習：作成問題４

（担当：佐々木晃）

演習問題B4
問題番号: B4
課題名:コンパイラの作成4

最終的には次のような言語のコンパイラを作成することが目的である。目的機械は、hsm 仮想機械とする。過去の講義資料（中田先生、開先生による）も参考にすること。

**問題** JavaCC を用いて，文法４と下記の仕様をみたすプログラムを作成せよ．
（新規分）

- getint 文，putnl 文を処理できる．
- do while 文を処理できる．
- If 文を処理できる。（余力がある者は if-else も扱えるようにせよ）
- 比較演算を処理できる．

前回までの分

- 変数名に英字で始まる英数字の文字列を利用できる．
- putint 文が使える。
- 変数名の未定義，二重定義のエラーをユーザーに知らせる．
- 変数，整数をオペランドとし，四則演算，単項マイナス，括弧を含む式を右辺に持つ代入文を処理できる．

プログラムの提出は提出指針に従うこと。
http://cis.k.hosei.ac.jp/~asasaki/lectureCompiler/guideline.htm

文法４

```
<PROGRAM>::= <MAIN>
<MAIN>::= 'int' 'main' '(' ')' <BLOCK>
<BLOCK>::= '{' <INTDECLLIST> <STATEMENTLIST> '}'
<INTDECLLIST>::= empty
              | <INTDECLLIST> <INTDECL>
<INTDECL>::= 'int' <IDENTLIST> ';'
<IDENTLIST>::= <IDENT>
            | <IDENTLIST> ',' <IDENT>
<STATEMENTLIST>::= empty
                | <STATEMENTLIST> <STATEMENT>
<STATEMENT>::= <SUBSTITUTION> '=' <EXPRESSION> ';'
            | '{' <STATEMENTLIST> '}'
            | <IFPREFIX> <STATEMENT> <IFPOSTFIX>
            | 'do' <STATEMENT> 'while' '(' <LOGICALEXPRESSION> ')' ';'
            | 'putint' '(' <EXPRESSION> ')' ';'
            | 'getint' '(' <SUBSTITUTION> ')' ';'
            | 'putnl'';'
<IFPREFIX> ::= 'if' '(' <LOGICALEXPRESSION> ')'
<IFPOSTFIX> ::= empty
                | ‘else’ <STATEMENT>
<SUBSTITUTION>::= <IDENT>

<LOGICALEXPRESSION>::= <LOGICALFACTOR>

<LOGICALFACTOR>::= <EXPRESSION> '==' <EXPRESSION>
```

```
                | <EXPRESSION> '!=' <EXPRESSION>
                | <EXPRESSION> '>=' <EXPRESSION>
                | <EXPRESSION> '>' <EXPRESSION>
                | <EXPRESSION> '<=' <EXPRESSION>
                | <EXPRESSION> '<' <EXPRESSION>

<EXPRESSION>::= <TERM>
             | <EXPRESSION> '+' <TERM>
             | <EXPRESSION> '-' <TERM>
<TERM>::= <UNARY>
       | <TERM> '*' <UNARY>
       | <TERM> '/' <UNARY>
<UNARY>::= <FACTOR>
        | '-' <UNARY>
<FACTOR>::= <IDENT>
         | <NUMBER>
         | '(' <EXPRESSION> ')'
<IDENT>::=英字で始まる英数字の繰り返し文字列
<NUMBER>::=数字の１回以上の繰り返し文字列
<CHARACTER>::=表示可能な ascii 文字を「'」で囲ったもの
-----------------------------------------
putint  <EXPRESSION>の値を整数で出力
getint  <SUBSTITUTION>に割り当てられた場所にコンソールからの入力整数を
        代入する.
putnl   改行コードをコンソールに出力する
```

## 補足説明

コンパイラの作成問題４のヒント（昨年度の解説ページを改変)

### 文法に関するヒント

JavaCC を用いる場合は， LL(1)となるように左括りだしをするか，LL(2)であるといった指定をする必要がある．たとえば以下のようにしてもよい．

```
<LOGICALFACTOR>:== <EXPRESSION>
                    ( '!=' <EXPRESSION> | '==' <EXPRESSION> | '>=' <EXPRESSION> | '>' <EXPRESSION> | '<=;'
<EXPRESSION> | '<' <EXPRESSION> )
```

また、左再帰性を取り除く必要がある.
下記は、作成問題３の範囲に関して、左再帰性を除いた例である。

```
<IDENTLIST>::= <IDENT> ( ',' <IDENT> )*
<INTDECLLIST>::= ( <INTDECL> )*
<SUBSTITUTION>::= <IDENT>
<STATEMENTLIST>::= ( <STATEMENT> )*
<EXPRESSION>::= <TERM> ( '+' <TERM> | '-' <TERM> )*
<TERM>::= <UNARY> ( '*' <UNARY> | '/' <UNARY> )*
<UNARY>::== <FACTOR>
        | '-' <UNARY>
<FACTOR>::=  <IDENT>
         | <NUMBER>
         | '(' <EXPRESSION> ')'
```

## 条件文(<LOGICALEXPRESSION>)における比較式

（以下において，<EXPRESSION1>～<EXPRESSION4>は，それぞれ数式を表す.）

<LOGICALEXPRESSION>が，<EXPRESSION> '>' <EXPRESSION>である場合を例に説明する.

（例 1）
　　<EXPRESSION1> '>' <EXPRESSION2>

は, <EXPRESSION1>の計算結果が, <EXPRESSION2>より大きければ，TRUE (1 で表現する) そうでなければ, FALSE (0 で表現する) がスタックのトップに 残るようにする必要がある. '>'に対応する命令は, GT である. この命令の意味は,

```
t=t-1; if s[t] > s[t+1] then s[t]=1 else s[t]=0;
```

なので，<EXPRESSION1>の計算結果の上に<EXPRESSION2>の計算結果が積まれたあとで， GT 命令を施せば，<EXPRESSION1>の計算結果が保持されていたアドレスに 1 または，0 が書き込まれた 状態がつくり出せる.
（例 1）に対応するコードは，以下のようになる.

```
　　（これ以前のコードでトップ（t）が第 k 段目に設定されているとする.）
　　<EXPRESSION1>に対応するコードの始まり
　　　　・
　　　　・
　　　　・
　　<EXPRESSION1>に対応するコードの終り　←第 k+1 段目に<EXPRESSION1>の結果が残る. t は，k+1 に設定される.
　　<EXPRESSION2>に対応するコードの始まり
　　　　・
　　　　・
　　　　・
　　<EXPRESSION2>に対応するコードの終り　←第 k+2 段目に<EXPRESSION2>の結果が残る. t は，k+2 に設定される.
　　GT 0 0　　←トップを一つ下げるので，t は，k+1 に設定され，そのアドレスには，1 または 0 が書き込まれている.
```

このように，<EXPRESSION1>に対応するコードを実行する直前と比べると，スタックが一段だけ高くなり， そこには，1 または 0 が書き込まれている状態がつくり出せる.
'==','!=','>=','<=','<'の場合も同様の考え方で処理できる.

## if 文(<IFPOSTFIX>が，empty の場合)

以下のようにその前後に<STATEMENT1>，<STATEMENT3>があるとして， <STATEMENT1>～<STATEMENT3>の実行順序を考えてみる． ただし，ここでは，<STATEMENT1>，<STATEMENT3>は，empty の場合もあるとする.

例 2)
```
　　<STATEMENT1>
　　if ( <LOGICALEXPRESSION> ) <STATEMENT2>
　　<STATEMENT3>
```

<LOGICALEXPRESSION>が TRUE であれば，<STATEMENT1>,<STATEMENT2>,<STATEMENT3> の順番で実行されるコードが必要とされ， <LOGICALEXPRESSION>が FALSE であれば，<STATEMENT1>,<STATEMENT3>の順番で実行される コードが必要とされている．すなわち，<LOGICALEXPRESSION>が FALSE の場合のみ<STATEMENT2>に 対応するコードを実行せずに（飛ばして,jump して）その次の<STATEMENT3>が実行されるようなコードを出力すればよい.
ここで必要となるような，「FALSE なら jump させる」命令が，FJ 命令である．この命令の意味は,

```
if s[t]==0 then pc=q; t=t-1;
```

すなわち，トップが FALSE ならば，トップを一つ下げて，次の命令は，第 q 行目から行なうことを意味している.
ここで，(例 2) に対応するコードを考えてみよう．まずは，今までのように，読み込んだ順番に命令コードを出力することを試みてみよう.

```
　　<STATEMENT1>に対応するコードの始まり
　　　　・
　　　　・
　　　　・
　　<STATEMENT1>に対応するコードの終り
　　<LOGICALEXPRESSION>に対応するコードの始まり
　　　　・
```

```
        ・
        ・
    <LOGICALEXPRESSION>に対応するコードの終り          ←この時点でトップに 1 または 0 が積まれている.
    FJ 0 q                                  ←この q オペランドの決定には，飛ばすべき<STATEMENT2>に対応するコードが必要である.
                                                しかし，<STATEMENT2>は，これから処理するので，この時点では，決定できない.
```

このように，飛び先は，これから決定されるコードの先にあるために，FJ 命令を出力するタイミングでは， 決定できないのである．必要とされているコードは，以下のようになる．

```
    <STATEMENT1>に対応するコードの始まり
        ・
        ・
        ・
    <STATEMENT1>に対応するコードの終り
    <LOGICALEXPRESSION>に対応するコードの始まり
        ・
        ・
        ・
    <LOGICALEXPRESSION>に対応するコードの終り      ←この時点でトップに 1 または 0 が積まれている.
    FJ 0 q                                  ←(A)
    <STATEMENT2>に対応するコードの始まり
        ・
        ・
        ・
    <STATEMENT2>に対応するコードの終り
    <STATEMENT3>に対応するコードの始まり  ←(A)の q オペランドは，この行の行番号
        ・
        ・
        ・
    <STATEMENT3>に対応するコードの終り
```

<STATEMENT2>に対応するコードが決まってから，(A)における FJ 命令の q オペランドを書き込む 必要があるため，今までのように標準出力やファイルに即座に出力するだけでは，必要とされるコードは得られない.

そこで全てのコードを配列などに保持しておき，ソースプログラムを全て読み込んだ後で， 標準出力やファイルに出力するようにすれば，FJ 命令が必要であることが判明するタイミングとそのオペランドが決定されるタイミングにずれがある場合にでも対応できる． また，配列を利用する以外に，リストなどを利用すれば，メモリが許す限り命令コードを保持できる．下記では，命令を Inst クラスであらわし、CodeTable クラスにコードを保持するようにするとする。CodeTable クラスの作成例は、最後に載せた。

```
class Inst{
    String operation; // 命令名
    int p;              // 第１引数（第１オペランド）
    int q;      // 第２引数（第２オペランド）
    Inst(String op, int a1, int a2){
      // コンストラクタ
        opCode = op;
        p = a1;
        q = a2;
    }

    public String toString(){
        return(operation+"¥t" + p + "¥t" + q);
    }
```

```
}
```

このような配列を用いた場合でも，<STATEMENT2>に対応するコードを 上記の配列に登録したあとで FJ 命令の q オペランドが決定されるので，そのオペランドを 書き込むためには，対応する FJ 命令がある行の行番号（配列の添字）を知っておく必要がある． この行番号の受渡しには JavaCC の場合は非終端記号の メソッドの返す値を使うとよい．

if 文に関係する文法を抜き出してみると以下の通りである．

```
<STATEMENT>::= <SUBSTITUTION> '=' <EXPRESSION> ';'
             | '{' <STATEMENTLIST> '}'
             | <IFPREFIX> <STATEMENT> <IFPOSTFIX>
 （一部省略）
<IFPREFIX>::= 'if' '(' <LOGICALEXPRESSION> ')'  (1)
<IFPOSTFIX>::= empty (2)
             | 'else' <STATEMENT>
```

if 文の構文解析中に(1)に到達した時点で<LOGICALEXPRESSION>のコードまでが登録されている． 次に登録されるのは，FJ 命令であり，この命令が登録された行番号を疑似変数を用いて(2)で利用する。

JavaCC では、非終端記号に対応するメソッドに引数を付けることが出来るので， <IFPOSTFIX>の引数として row の値を渡すことにすれば，<IFPOSTFIX>が empty であると分かった時点でその処理が出来る．たとえば以下のようにすればよい．

```
<STATEMENT>::= <SUBSTITUTION> '=' <EXPRESSION> ';'
             | '{' <STATEMENTLIST> '}'
             | row1=<IFPREFIX>() <STATEMENT> <IFPOSTFIX>(row1)

 （一部省略）

int <IFPREFIX>()::= 'if' '(' <LOGICALEXPRESSION> ')'
          {CodeTable の row 行目に Inst(FJ, 0, 0)を登録 (q オペランドの 0 はダミー、以降 Inst(FJ, 0, #)のように書く)；
           return row++;}

void <IFPOSTFIX>(int r)::= 'else' <STATEMENT>
             | /* empty */ {CodeTable の r 行目の q オペランドを row に変える。;}
```

ここで、「CodeTable の row 行目に Inst(FJ, 0, 0)を登録」とは、新たに命令 FJ 0 0 を CodeTable に追加することを意味する。(すなわち、row は新たな命令を追加する行番号であり、row-1 行まではすでにそれまでの解析によってコードが保持されている状態。) 最後の CodeTable クラスのメソッドでは、ct を CodeTable のインスタンスとして、ct.addCode(FJ, 0, 0);となる。row++などは、addCode を読んだときにメソッドの内で行うのが良い。
「CodeTable の r 行目の q オペランドを row に変える」は、ct.backpatch(r, row);に対応する。

### if else 文(<IFPOSTFIX>が, 'else' <STATEMENT>の場合)

if 文と同様にその前後に<STATEMENT1>, <STATEMENT4>があるとして， <STATEMENT1>～<STATEMENT4>の実行順序を考えてみる． ただし，ここでは，<STATEMENT1>, <STATEMENT4>は，empty の場合もあるとする．

```
      （例 3）
     <STATEMENT1>
     if ( <LOGICALEXPRESSION> ) <STATEMENT2> else <STATEMENT3>
     <STATEMENT4>
```

<LOGICALEXPRESSION>が TRUE であれば，<STATEMENT1>,<STATEMENT2>,<STATEMENT4> の順番で実行されるコードが必要とされ， <LOGICALEXPRESSION> が FALSE であれば，<STATEMENT1>,<STATEMENT3>,<STATEMENT4>の順番で実行されるコードが 必要とされる．すなわち，

```
     <STATEMENT1>に対応するコードの始まり
             ・
             ・
```

```
        ・
    <STATEMENT1>に対応するコードの終り
    <LOGICALEXPRESSION>に対応するコードの始まり
        ・
        ・
        ・
    <LOGICALEXPRESSION>に対応するコードの終り      ←この時点でトップに 1 または 0 が積まれている.
    FJ 0 q                                  ←(A) FJ 命令が必要となった時点では, ここの q は, 未決定.
    <STATEMENT2>に対応するコードの始まり
        ・
        ・
        ・
    <STATEMENT2>に対応するコードの終り
    J  0 q                                  ←(B) J 命令が必要となった時点では, ここの q は, 未決定.
    <STATEMENT3>に対応するコードの始まり  ←この時点で, (A)の q オペランドが, この行の行番号に決定される
        ・
        ・
        ・
    <STATEMENT3>に対応するコードの終り
    <STATEMENT4>に対応するコードの始まり  ←この時点で, (B)の q オペランドが, この行の行番号に決定される
        ・
        ・
        ・
    <STATEMENT4>に対応するコードの終り
```

このように, <LOGICALEXPRESSION>が TRUE の時に<STATEMENT3>が必ず実行されない (飛ばされる, jump される) ためには, J 命令を利用するとよい. J 命令の意味は,

```
pc=q;
```

なので, 無条件に次の命令は, 第 q 行目から行なうことを意味している. (A),(B)の q オペランドは, if 文の時と同じ理由により, その先のコードが処理されない限り決定されない.

```
<STATEMENT>::= <SUBSTITUTION> '=' <EXPRESSION> ';'
            | '{' <STATEMENTLIST> '}'
            | row1=<IFPREFIX>() <STATEMENT> <IFPOSTFIX>(row1)

 (一部省略)

int <IFPREFIX>()::= 'if' '(' <LOGICALEXPRESSION> ')'
          {CodeTable の row 行目に Inst(FJ, 0, #)を登録;
           return row++;}
int <IFPOSTFIX>(int row1){
     'else'
       {CodeTable の row 行目を Inst(J, 0, #)とする;
          row2=row;row++;
          CodeTable の row1 行目の q オペランドを row に;}
      <STATEMENT>
        {CodeTable の row2 番目の q オペランドを=row に;}  // ここまでが else 節がある場合の処理。
        | /* empty */ {CodeTable の row1 行目の q オペランドを row に;}
```

**shift/reduce conflict について**

if 文と if else 文では, shift(生成)／reduce(還元)の conflict(衝突)が発生しワーニングが一つ出る. javaCC では先に書いたものが優先されるので, 次のように書けばよい. (ε(empty)は常に最後に書かなければならない. また「{}」も付けておく必要がある.)

```
        void ifpostfix() : {}
        { "else" statement()
         | {}
        }
```

LR 構文解析を扱える yacc(bison)などのツールでは，shift が優先されるため，既に現れた'(' <LOGICALEXPRESSION> ')' の中で，まだどの else とも 組み合わされていない'(' <LOGICALEXPRESSION> ')'の中で，その else に最も近いものと組み合わされる．

## while 文

以下のようにその前後に<STATEMENT1>，<STATEMENT3>があるとして， <STATEMENT1>～<STATEMENT3>の実行順序を考えてみる． ただし，ここでは，<STATEMENT1>，<STATEMENT3>は，empty の場合もあるとする．

```
（例 4）
<STATEMENT1>
while ( <LOGICALEXPRESSION> ) <STATEMENT2>
<STATEMENT3>
```

<STATEMENT1>が実行された後に，<LOGICALEXPRESSION>が TRUE である間は，<STATEMENT2>が 繰り返し実行され，<LOGICALEXPRESSION>が FALSE になると<STATEMENT3>が実行されるコードが 必要となる．
<LOGICALEXPRESSION>が TRUE である間，<STATEMENT2>が繰り返し実行されるようにするには，<STATEMENT2>に対応するコードの次に J 命令を登録し，その際の飛び先（q オペランド）を，<LOGICALEXPRESSION>に対応するコードの始まりが登録されている行番号にすればよい．すなわち，

```
<STATEMENT1>に対応するコードの始まり
        ・
        ・
        ・
<STATEMENT1>に対応するコードの終り
<LOGICALEXPRESSION>に対応するコードの始まり    ←(C)
        ・
        ・
        ・
<LOGICALEXPRESSION>に対応するコードの終り      ←この時点でトップに 1 または 0 が積まれている．
FJ 0 q                                  ←(A) FJ 命令が必要となった時点では，ここの q は，未決定．
<STATEMENT2>に対応するコードの始まり
        ・
        ・
        ・
<STATEMENT2>に対応するコードの終り
J  0 (C)の行番号                        ←(D) J 命令が必要となった時点で，(C)の行番号は，既に決定．
<STATEMENT3>に対応するコードの始まり  ←この時点で，(A)の q オペランドが，この行の行番号に決定される
        ・
        ・
        ・
<STATEMENT3>に対応するコードの終り
```

while 文に関係する文法を抜き出してみると以下の通りである．

```
<STATEMENT>::= <SUBSTITUTION> '=' <EXPRESSION> ';'
（一部省略）
        | 'while' '(' <LOGICALEXPRESSION> ')' <STATEMENT>
（一部省略）
```

<LOGICALEXPRESSION>の直前で，その時点での row が<STATEMENT>の後ろのアクションで利用できる ように疑似変数に代入し，<LOGICALEXPRESSION>の直後で FJ 命令を，<STATEMENT>の直後で J 命令を 登録すれば，必要とされるコードを得られる．まとめると，

```
<STATEMENT>::= <SUBSTITUTION> '=' <EXPRESSION> ';'
```

```
（一部省略）
          | 'while' '(' {row1=row;} <LOGICALEXPRESSION>
            {CodeTable の row 行目を Inst(FJ, 0, #)とする;
             row2=row;row++;} ')'
            <STATEMENT> {CodeTable の row 行目を Inst(J, 0, row1)とする;
                         row++;
                           CodeTable の row2 行目の q オペランドを row とする;}

（一部省略）
```

## do while 文

以下のようにその前後に<STATEMENT1>, <STATEMENT3>があるとして， <STATEMENT1>～<STATEMENT3>の実行順序を考えてみる． ただし，ここでは，<STATEMENT1>，<STATEMENT3>は，empty の場合もあるとする．

```
（例 5）
<STATEMENT1>
do
<STATEMENT2>
while ( <LOGICALEXPRESSION> )
<STATEMENT3>
```

<STATEMENT1>が実行された後に，<STATEMENT2>が実行され，その後，<LOGICALEXPRESSION>が TRUE である間は，<STATEMENT2>が 繰り返し実行され，<LOGICALEXPRESSION>が FALSE になると<STATEMENT3>が実行されるコードが必要となる．

<LOGICALEXPRESSION>が TRUE である間，<STATEMENT2>が繰り返し実行されるようにするには，<STATEMENT2>に対応するコードの次に<LOGICALEXPRESSION>に対応するコードが始まり，その結果が TRUE であれば， <STATEMENT2>に対応するコードの始まりにジャンプすればよい．

TRUE であれば，ジャンプするので，仮に，TJ 命令(True Jump:, if s[t]==1 then pc=q else pc=pc+1;t=t-1;)が命令コードにあるのものとして， 説明する．hsm には，そのような命令は，用意されていないので，同様の機能を考え出すこと．

この TJ 命令を用いて，<LOGICALEXPRESSION>が TRUE である間，<STATEMENT2>が繰り返し実行されるようにするには， <LOGICALEXPRESSION>に対応するコードの次に TJ 命令を登録し，その際の飛び先（q オペランド）を，<STATEMENT2>に対応するコードの始まりが登録されている行番号にすればよい．すなわち，

```
<STATEMENT1>に対応するコードの始まり
        ・
        ・
        ・
<STATEMENT1>に対応するコードの終り
<STATEMENT2>に対応するコードの始まり     ←(A)この時点で，do を認識している．
        ・
        ・
        ・
<STATEMENT2>に対応するコードの終り
<LOGICALEXPRESSION>に対応するコードの始まり
        ・
        ・
        ・
<LOGICALEXPRESSION>に対応するコードの終り       ←この時点でトップに 1 または 0 が積まれている．
TJ 0 (A)の行番号                          ←(B) TJ 命令が必要となった時点で，必要な情報は，全て揃っている．
<STATEMENT3>に対応するコードの始まり
        ・
        ・
        ・
<STATEMENT3>に対応するコードの終り
```

do while 文に関係する文法を抜き出してみると以下の通りである．

```
<STATEMENT>::= <SUBSTITUTION> '=' <EXPRESSION> ';'
```

```
 (一部省略)
          | 'do' <STATEMENT> 'while' '(' <LOGICALEXPRESSION> ')'
 (一部省略)
```

'do'の直後で,その時点でのrowが<LOGICALEXPRESSION>の後ろのアクションで利用できる ように疑似変数に代入し,<LOGICALEXPRESSION>の直後でTJ命令を，登録すれば，必要とされるコードを得られる．まとめると，

```
<STATEMENT>::= <SUBSTITUTION> '=' <EXPRESSION> ';'
 (一部省略)
          | 'do' {row1=row;} <STATEMENT> 'while' '(' <LOGICALEXPRESSION>
                              {CodeTableのrow行目をInst(TJ, 0, row1とする);row++;} ')'
 (一部省略)
```

.

## コードテーブルの例(addCode, backPatch, listCodeは未完)

```
class CodeTable{
    Inst[] codeTable;  // 命令語を配列に保持する。
    int index;  //最後に生成した命令語のインデックス
    int size;   //最大命令長
    CodeTable(int s){
        // コンストラクタ
        // 最大命令長を初期値sとして設定する。
        size = s;
        index = -1;
        codeTable = new Inst[size];
    }

    int addCode(String opCode, int p, int q) throws RuntimeException {
        // 命令をテーブルに追加する。
        // その命令のindexの値を返す。
        // 命令を追加できない場合は例外を投げる。
    }

    int getIndex(){
      return index;
    }

    void backPatch(int ind, int target){
        //codeTable[ind]のとび先をtargetとする。
    }

    public void listCode(int debug){
        // 目的コードの表示
    }
}
```

Ｉｎｓｔクラス(再掲)

```
class Inst{
    String operation; // 命令名
    int p;              // 第１引数（第１オペランド）
    int q;     // 第２引数（第２オペランド）
    Inst(String op, int a1, int a2){
```

```
      // コンストラクタ
         opCode = op;
         p = a1;
         q = a2;
    }

    public String toString(){
         return(operation+"¥t" + p + "¥t" + q);
    }
}
```

NameTable, CodeTable を使う場合には jj ファイルは次のような構成になるだろう。

```
PARSER_BEGIN(Compiler4)
public class Compiler4 {
  static NameTable nameTable;
  static CodeTable codeTable;
  public static void main(String args[]) {...}
}
class Name{...}
class NameTable{...}
class Inst{
....
}
class CodeTable{
...
}
PARSER_END(Compiler4)
```