コンパイラ演習：作成問題 3

（担当：佐々木晃）

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。昨年度までの講義資料（中田先生、開先生による）も参考にすること。

演習問題 B3
問題番号: B3
課題名:コンパイラの作成 3
(1) 記号表の実装
(2) JavaCC プログラム課題 3

（1）記号表の実装

記号表を実現するクラス NameTable を作成し、動作を確かめよ。記号表に登録する名前（エントリ）はクラス Name として、「つづり」と「登録したアドレス」を情報として持つこととする。その他の機能は例を参考にせよ。（ただし、メソッド本体は、省略している。）

ソースプログラムのアップロードは不要です。プログラムはレポートに添付してください。
実行例や、内容の説明は必要です。

（2）JavaCC プログラム課題 3

次のような機能を持つ言語のコンパイラを作成せよ。文法は前回と同じ。

文法：

```
<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> ';'
            | 'putint' '(' <EXPRESSION> ')' ';'
<SUBSTITUTION>::= <IDENT>
<EXPRESSION>::= <TERM>
             | <EXPRESSION> '+' <TERM>
             | <EXPRESSION> '-' <TERM>
<TERM>::= <UNARY>
       | <TERM> '*' <UNARY>
       | <TERM> '/' <UNARY>
<UNARY>::== <FACTOR>
        | '-' <UNARY>
<FACTOR>::= <IDENT>
         | <NUMBER>
         | '(' <EXPRESSION> ')'
```

字句の定義（終端記号）
o 空白、タブ、改行（これらは構文の中では使われない。スキップする）

o <IDENT>::=英字で始まる英数字の繰り返し文字列
o <NUMBER>::=数字の１回以上の繰り返し文字列
o 括弧記号、区切り記号など
o 演算記号
o その他キーワード（予約語） int, main, putint など

（empty はε（空の記号列）を意味する。）

---

putint <EXPRESSION>の値を整数で出力

例：

```
int main(){
  int a;
  int b;
  a = 1;
  b = 2;
}
```

結果：

```
PUSH 0 2
LDC 0 1
STV 0 0
LDC 0 2
STV 0 1
POP 0 2
HLT 0 0
```

メモ：紙レポートに入れるべきもの。
○JavaCC のソースプログラム。（別途プログラムは指定場所にアップロードすること）
○実行結果。
(1) ソースプログラム
実行結果は、求められている機能がひととおり実装できたことを証明するものであるから、それに足る例題（ソースプログラム）をいくつか用意すること。
(2) ソースプログラムの翻訳結果（hsm コード、エラーの場合はエラーメッセージ）。
(3) 翻訳結果を hsm 上で実行させたときのスナップショット。(putint による出力がない場合など、最後まで実行させると、スタックの中身がなくなり本当に実行しているかわからない場合は、POP する手前の結果が良いだろう。)

○ いつもどおり考察は必要です。

補足説明（作成問題３）

下記のプログラムにおける記号表を考えてみよう。

```
int main()
{
  int abc;
  int cd;
  int efg;
  cd = 10;
  putint(cd);
}
```

このプログラムでは、変数”abc”,”cd”,”efg”の登録と、”cd”の検索を行っている。この動作に従って、次のような確認を手動で行ってみよ。(参考例の main メソッドを参照。)

1. 変数"abc","cd", "efg"を登録する。

2. 1の状態での記号表の内容を出力せよ。
3. 変数"cd"を表から検索して、内容を表示せよ。

動作例：

```
id:abc, address:0    ←  記号表の内容を表示（上記2）
id:cd, address:1
id:efg, address:2

id:cd, address:1    ←  変数cdに対するエントリの検索結果(上記3)
```

上記の動作確認のほか、２重宣言のエラーや、未宣言の変数の式での使用のエラーとなる例も、確認しこれをレポートに載せて下さい。

記号表のプログラムの例(メソッドの中身は一部省略)：

NameTable.java

```
class Name{
    String ident;
    int type;
    int address;
    Name(String id, int a){
        // コンストラクタ
    }
    public String toString(){
        // エントリの内容の文字列表現を返す。
        // 例  id:abc, address:3
    }
}

public class NameTable{
    Name[] nameTable; // テーブル本体。名前（エントリ）の配列
    int index;
    int nextAddress; // 次に宣言される変数のaddressの値
    int maxIndex;
    NameTable(int max){
        // コンストラクタ。テーブルに登録できるエントリ数はmaxとする。
    }

    int addName(String ident) throws RuntimeException{
        // テーブルに名前（エントリ）を登録する。エラーがあったら例外を投げる
        // 領域のどこに登録したかを返す。
    }

    Name searchName(String ident) throws RuntimeException{
        // テーブルにつづりidentをもつ名前（エントリ）を探索し、あればそれを返す。
        // エラーがあったら例外を投げる
    }

    int getNextAddress(){
        // nextAddressを返す。
    }

    public String toString(){
        // nameTableの内容の文字列表現を返す。
        // 例：
        //  id:abc, address:0
```

```
        //  id:c, address:1
        //  id:def, address:2
        //  (ネームテーブルをプリントするときのため)
        // 下記は実装例
        StringBuffer buf = new StringBuffer();
        for (int i=0; i<index; i++){
            buf.append(nameTable[i].toString());
            buf.append("¥n");
        }
        return buf.toString();
    }

    public static void main(String arg[]){
        NameTable nt = new NameTable(256);
        nt.addName("abc");
        nt.addName("cd");
        nt.addName("efg");
        System.out.println(nt);
        // println(#)は、内部で#.toString()を呼ぶことに注意。
        Name entry = nt.searchName("cd");
        System.out.println(entry);
    }
}
```

- 例外を投げるときは、throw new RuntimeException("エラーメッセージ");のようにすればよい。Java II などの講義資料を参考にすること。

補足説明（作成問題３）

コンパイラの作成問題３のヒント（昨年度の解説ページを改変）

変数名に対するアドレスとして，スタックの何段目を利用させるかの管理には，変数表（記号表、ネームテーブル）を作成する必要がある．コンパイラの内部では，構造体の配列などによるテーブルを用いて変数表を実装する．例えば，以下のようなクラスを用意すると良い。（本日の問題（１）と同じものです。Main メソッドがないだけです。）

```
class Name{
    String ident;
    int address;
    Name(String id, int a){
        // コンストラクタ
    }
    public String toString(){
        // エントリの内容の文字列表現を返す。
        // 例  id:abc, address:3
    }
}

public class NameTable{
    Name[] nameTable; // テーブル本体。名前（エントリ）の配列
    int index;
    int nextAddress; // 次に宣言される変数の address の値
    int maxIndex;
    NameTable(int max){
        // コンストラクタ。テーブルに登録できるエントリ数は max とする。
```

```
    }

    int addName(String ident) throws RuntimeException{
        // テーブルに名前（エントリ）を登録する。エラーがあったら例外を投げる
        // 領域のどこに登録したかを返す。
    }

    Name searchName(String ident) throws RuntimeException{
        // テーブルにつづり ident をもつ名前（エントリ）を探索し、あればそれを返す。
        // エラーがあったら例外を投げる
    }

    int getNextAddress(){
        // nextAddress を返す。
    }

    public String toString(){
        // nameTable の内容の文字列表現を返す。
        // 例：
        //  id:abc, address:0
        //  id:c, address:1
        //  id:def, address:2
        //  （ネームテーブルをプリントするときのため）
        // 下記は実装例
        StringBuilder buf = new StringBuilder();
        for (int i=0; i<index; i++){
            buf.append(nameTable[i].toString());
            buf.append("¥n");
        }
        return buf.toString();
    }
}
```

- Name クラスの ident には，変数名を保持しておく．(token.image を利用する)
- address には，base からの相対アドレスを保持する．（今回は，base は，0 です．）
- 例外を投げるときは、throw new RuntimeException("エラーメッセージ");のようにすればよい。

宣言文で必要とされる処理は，上記のテーブルに必要なデータを登録することである．データを登録する前に，同じ名前が既に登録されているか調べ，もし登録されていれば，二重定義の処理をして終了する．登録処理の概要は，以下の通りである．

- nameTable の index 番目を利用する変数を登録する場合（index の初期値は，0 とする）
  nameTable[0]～nameTable[index-1]までに今から登録する名前と同じ identifier が
  あるかチェックし，あれば（二重定義されていれば），然るべき処理をして終了．
  そうでなければ，
  Name の新しいインスタンスを作り(ここでは n とする)、nameTable[index]に登録する。
  n に登録する内容等は下記のとおり。
  n.ident へ文字列代入;
  n.address=nextAddress;
  nextAddress=n.address+1;
  index++;

今回は，変数領域の確保として，

PUSH 0 nt.getNextAddress();

および，HLT 命令の直前で，

POP 0 nt.getNextAddress();

が必要となる．　ここで nt は名前表、すなわち NameTable のインスタンスである。

参照・代入で必要とされる処理は，必要とされる変数が利用する段（アドレス）を計算して求めることである．

1. 字句解析が検出した文字列と等しい identifier を持つテーブル要素を見つけ出す．
   もし見付けられなかったら，変数または配列の未定義の処理をして終了する．
2. 見つけ出した変数の address を，LDV 命令で利用する．
3. putint 文,getint 文,の処理でアドレスの決定が必要な場合，上記と同じ方法が利用できる．

JavaCC の記述に関するヒント

<SUBSTITUTION>について

代入文では、代入の左辺に非終端記号<SUBSTITUTION>を導入して、下記のようにしている。これは、将来配列要素への（たとえば a[10]=...）代入を行うためである。

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

さて、代入文に対するコードは、上記のとおり STV 命令を用いて次のようになるはずである。

```
void Statement() {}
{
   Substitution() '=' Expression() { STV 命令の出力 }
  | ...
}
```

ところが、変数に対する番地は substitution の<IDENT>の文字を読まなければ計算できない。

```
void Substition() {Token tk;}
{
        tk=<IDENT> { tk.image ???; }
}
```

javacc では、非終端記号で解析した結果をメソッドの返り値として、情報を（解析木の上のほうに）渡すことができる。（下のほうに渡す場合はメソッドのパラメータを用いる）したがって、下記のようにするとよい。

```
void Statement() {String st;}
{
   st=Substitution() '=' Expression() { STV 命令の出力（st を使って STV の番地を計算が可能）; }//(1)
  | ...
}

String Substition() {Token tk;}
{
        tk=<IDENT> { return tk.image; } //(2)
}
```

この場合、Substitution()というメソッドの中で終端記号<IDENT>の文字列が求められるので、この文字列を return で返すようにする(2)。(1)の st=Substitution()で、Substitution()で求めた文字列（=<IDENT>の返り値）が変数 st に代入される。

もちろん、今回の場合、

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

を

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

とすれば、問題は生じない。

プログラムについての補足

JavaCC の中に、クラスを記述するには下記のようにする。パッケージとしてクラス等を作成する場合は、適宜メソッド、フィールド等を public 宣言する必要などがある。

```
PARSER_BEGIN(Compiler3)
import java.io.*;
public class Compiler3 {
  // ....
  static NameTable nameTable;
  public static void main(String args[]) {
     nameTable = new NameTable(256);
     ...
  }
}

class Name{
...
}

class NameTable{
  ....

}
PARSER_END(Compiler3)
```