コンパイラ演習：作成問題 2

（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。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 B2
問題番号: B2
課題名：コンパイラの作成２
問題：
(1)ハンドコンパイル問題集
(2) JavaCC プログラム課題

(1) ハンドコンパイル問題集

下記のプログラムに対する hsm コードを示し、実際に hsm での動作を確認せよ。

○動作の確認は、コンパイルエラーとならない例のみでよい。

○コンパイルする際に使用した記号表も示すこと。なお、記号表への変数の登録時や、コードの変換の途中でコンパイルエラーとなる場合は、そのエラーが出た時での記号表や翻訳の途中結果がどうなっているかを示せ。

(a)

```
int main(){
  int a, b;
  b = 10;
  putint(b);
}
```

(b)

```
int main(){
  int a, b;
  int a;
  b = 10;
  putint(b);
}
```

(c)

```
int main(){
  int b,c;
  b = 10;
  putint(a);
}
```

(d)

```
int main(){
  int a,b;
  c = 10;
  putint(a);
}
```

(e)

```
int main(){
  int a,b;
  a = 2;
  b = 10;
  b = a + b * 2;
  putint(a);
  putint(b);
}
```

(f)

```
int main(){
  int a,b;
  int c,d;
  a = 2;
  b = a;
  c = b * b + a;
  putint(c);
}
```

(2) JavaCC プログラム課題

次のような機能を持つ言語のコンパイラを作成せよ。今回は hsm コードではなく、講義資料にあるような hsm の擬似コードに翻訳するようにせよ。(すなわち、コンパイル結果はそのままでは hsm では動かない。)

機能：

- 変数名に英字で始まる英数字の文字列を利用できる.
- 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.
- putint 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。(ただし、今回は hsm での実行は試さなくて良い。)
http://cis.k.hosei.ac.jp/~asasaki/lectureCompiler/guideline.htm

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;
}
```

次のような hsm の命令風のコードを出力するようにせよ。

```
DECL a b  -> 本来はこの命令はない。
PUSH 0 2  (-> 確保する領域の個数を指定する。)
LDC 0 1
STV 0 a -> 本来は STV 0 0
LDC 0 2
STV 0 b  ->本来は STV 0 1
POP 0 2   (-> 確保したメモリの開放。)
HLT 0 0
```

なお、PUSH 命令、POP 命令も JavaCC の練習のために含めること。実際は、擬似コードではこれらの命令は本来あまり意味がない。（なので、資料の「擬似 hsm コード」には含めていない）

このコードは当然そのままでは実行できないので、記号表を各自作成して、それに基づいて変数を実際のアドレスに手で置き換えた（テキストエディタなどで書き換えた）ものを hsm で実際に動かしてみよ。

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。
　ソースプログラムは（1）のハンドコンパイルドリルのもの(a)-(f)の 6 つを用いる。（それ以上であるのは OK）
　(A) 作成したコンパイラの翻訳結果（擬似 hsm コード）。
　(B) 翻訳結果（疑似 hsm コード）を手で置き換え、hsm 上で実行させたときのスナップショット。（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ので、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。

口補足：

○変数宣言の翻訳

```
DECL a b
PUSH 0 2
LDC 0 1
....
```

の部分は、次のように表示を工夫してもよい。

```
宣言される変数は a, b の 2 個です。
PUSH 0 2
LDC 0 1
```

```
...
```

○変数におけるエラーについて

```
int main(){
  int a;
  int a;
}
```

のような場合、本来二重定義のエラーだが、今回はチェックせずに、

```
DECL a a
PUSH 0 2
POP 0 2
HT 0 0
```

あるいは

```
DECL a
DECL a
PUSH 0 2
POP 0 2
HT 0 0
```

でよい。

同様に、

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

や、

```
int main(){
  putint(a)
}
```

は未定義変数の使用のエラーだが、チェックはしなくてよい。それぞれ、

```
DECL a
PUSH 0 1
LDC 0 2
STV 0 b
POP 0 0
HLT 0 0
```

```
DECL
PUSH 0 0
LDV 0 a
WRI 0 0
POP 0 0
HLT 0 0
```

のように翻訳できていればOK.
余力のある人は、エラーのチェックをしてももちろん良い。

JavaCCの記述に関するヒント

ヒント：文法はJavaCCで書けるように、左再帰を除去する必要がある。
拡張バッカス記法（正規右辺文法）の形に直すのがよい。変数宣言の部分

```
<IDENTLIST>::= <IDENT>
```

```
            | <IDENTLIST> ',' <IDENT>
```
は、
```
<IDENTLIST> ::= <IDENT> {',' <IDENT>}
```
となる。JavaCC で文法記号列の 0 回以上の繰り返しは、それらを(…)*で囲えばよい。

ヒント：宣言された変数の数を数える必要があるが、それをカウントしておくための変数(numDeclared)を次のように定義するとよい。
```
PARSER_BEGIN(Compiler3)
import java.io.*;
public class Compiler3 {
  // ....
  static int numDecalred;  // 宣言された変数の数
  public static void main(String args[]) {
     numDeclared = 0;
     ...
  }
}
PARSER_END(Compiler3)

// コンパイラ本体
```

宣言部について

宣言の部分では、アクション（下記[]内の処理）は、
```
<IDENTLIST>::= <IDENT> [put(<IDENT>); numDeclared++;]
            | <IDENTLIST> ',' <IDENT> [put(<IDENT>); numDeclared++;]
```
のようにすればよい。

拡張バッカス記法による文法
```
<IDENTLIST> ::= <IDENT> {',' <IDENT>}
```
であれば、
```
<IDENTLIST> ::= <IDENT>  [put(<IDENT>); numDeclared++;]
                {',' <IDENT>   [put(<IDENT>); numDeclared++;]}
```
2 行目は「',' を読み <IDENT> を読み込む。次に[] の中身を実行する」を n 回(0 以上)繰り返すこと、を意味する。

さらに javacc の文法に近づけて書くと、ローカル変数 token を導入して、次のようになるであろう。4 行目の括弧内が繰り返し部分の処理となる。
```
void <IDENTLIST> () {Token token;}
{
   token=<IDENT>  {put(token.image + " "); numDeclared++;}
   (',' token=<IDENT> {put(token.image + " "); numDeclared++;})*
}
```

<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。