コンパイラ演習：作成問題 5

（担当：佐々木晃）

目的機械は、hsm 仮想機械とする。

問題番号: B5
課題名:コンパイラの作成 5

問題 JavaCC を用いて，文法 5 と下記の仕様をみたすプログラムを作成せよ.
斜体の部分(配列機能、コメント機能、論理演算)は余力のある人向けです.

（追加分）

- 関数を処理できる.
- 関数名に英字で始まる英数字の文字列を利用できる.
- 関数の未定義，二重定義，引数の個数の不一致のエラーをユーザーに知らせる.
- 大域変数を処理できる.
- 大域の変数名，*配列名*に英字で始まる英数字の文字列を利用できる.
- 大域の変数，*配列*は，全部で 256 個利用できる.
- 大域の変数，*配列*の未定義，不一致，二重定義のエラーをユーザーに知らせる.

　　不一致とは、int で宣言された変数を配列として使おうとする、関数として宣言された変数に int の値を代入しようとするなど。

- */*から*/はコンパイルの処理は行わない．(読み飛ばす．コメントの処理)*
- *論理演算のサポート。ただし，Java やC とは違い，カッコとして[]を使い，優先順位は,*
  (高)　　>=, >, <=, <, !=, ==　→　!　→　&&　→　^, //　(低)
  とする。→前回の ppt 資料や 2008 年度前期の第 4 回課題の資料を参考にするとよい。

（変更分）

- 関数，変数，*配列要素*，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.

（前回分）

- putnl 文を処理できる.
- getint 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.

（前々回分まで）

- 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５ (斜体部（配列）は余力のある人向け)

```
<PROGRAM>::= <DECLS> <MAIN>
<DECLS>::= empty
        | <DECLS> <INTDECL>
        | <DECLS> <FUNCDECL>
<INTDECL>::= 'int' <IDENTLIST> ';'
<FUNCDECL>::= 'int' <IDENT> '(' <FORMALLIST> ')' <BLOCK>
<FORMALLIST>::= empty
             | <IDENTLIST>
```

```
<IDENTLIST>::= <IDENT>
            / <IDENT> '[' <NUMBER> ']'
            | <IDENTLIST> ',' <IDENT>
            / <IDENTLIST> ',' <IDENT> '[' <NUMBER> ']'
<MAIN>::= 'int' 'main'  '(' ')' <BLOCK>
<BLOCK>::= '{' <INTDECLLIST> <STATEMENTLIST> '}'
<INTDECLLIST>::= empty
              | <INTDECLLIST> <INTDECL>
<STATEMENTLIST>::=empty
                | <STATEMENTLIST> <STATEMENT>
<STATEMENT>::= <SUBSTITUTION> '=' <EXPRESSION> ';'
            | '{' <STATEMENTLIST> '}'
            | 'if' '(' <LOGICALEXPRESSION> ')'   <STATEMENT>
            | 'do' <STATEMENT> 'while' '(' <LOGICALEXPRESSION>  ')'
            | 'return' <EXPRESSION> ';'
            | 'putint' '(' <EXPRESSION> ')' ';'
            | 'getint' '(' <SUBSTITUTION> ')' ';'
            | 'putnl'';'
<SUBSTITUTION>::= <IDENT>
              / <IDENT> '[' <EXPRESSION> ']'

<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>
         / <IDENT> '[' <EXPRESSION> ']'
         | <NUMBER>
         | <IDENT> '(' <ACTUALLIST> ')'
         | '(' <EXPRESSION> ')'
<ACTUALLIST>::= empty
             | <EXPRESSIONLIST>
<EXPRESSIONLIST>::= <EXPRESSION>
                 | <EXPRESSIONLIST> ',' <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

-------------------------------------
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

論理演算の文法（余力のある人向け）：上記文法の<LOGICALEXPRESSION>以下の部分が、次のように変更となる。

```
<LOGICALEXPRESSION>::= <LOGICALTERM>
                     | <LOGICALEXPRESSION> '||' <LOGICALTERM>
                     | <LOGICALEXPRESSION> '^^' <LOGICALTERM>
<LOGICALTERM>::= <LOGICALUNARY>
               | <LOGICALTERM> '&&' <LOGICALUNARY>
<LOGICALUNARY>::= <LOGICALFACTOR>
                | '!' <LOGICALUNARY>
<LOGICALFACTOR>::= <EXPRESSION> '!=' <EXPRESSION>
                 | <EXPRESSION> '>=' <EXPRESSION>
                 | <EXPRESSION> '>' <EXPRESSION>
                 | <EXPRESSION> '<=' <EXPRESSION>
                 | <EXPRESSION> '<' <EXPRESSION>
                 | '[' <LOGICALEXPRESSION> ']'
```

補足説明

コンパイラの作成問題５のヒント

次の段階が考えられる。この課題に関しては、A,B の段階に応じて評価を行うことにする。C は配列ができた人向け。

A: 引数なしの関数宣言と呼び出し。
B: 引数ありの関数宣言と呼び出し。(引数は int)
C(余力のある者): 引数ありの関数宣言と呼び出し。(B+配列の参照呼び)（完成形)

指針:

A: 引数なしの関数宣言と呼び出し

前段階として、(1)大域変数宣言とその参照, (2)main 関数の宣言、呼び出しができるようにすることを目指すと良い。

```
(test-a0.h)
int main(){
  int a;
  int b;
  a = 1;
  b = 2;
  putint(a+b);
}

(test-a1.h)
int g;
int main(){
  int a;
  int b;
  a = 1;
  b = 2;
  g = 3;
  putint(a);
  putint(g);
  return(0);
}
```

文法について

まず、大域変数の宣言を加えた場合を考える。

```
int i;
int main(){
  ..
}
```

この場合、int i;　の宣言が大域変数の宣言なのか、メインプログラムの宣言なのかは、int の次のトークンまで読まないと判別がつかないので、LOOKAHEAD(2)とするとよい。

```
DECLS : {}
{
  (LOOKAHEAD(2) INTDECL)*
}
```

なお関数宣言を加える場合には、下記を考えると、

```
int i;
int f(x){
..}
int main(x){
...
}
```

int i の i が変数宣言か関数宣言なのかは、その次のトークンが';'か'('で決まるため、括り出し等を用いない場合は、LOOKAHEAD(3)が必要になる。

INTDECL, FUNCDECL をまとめて考えたときの JavaCC のコードの例は次の形になる。

```
void <PROGRAM>() : {}
{
  <DECLS>(); <MAIN>();
}

void <DECLS>() : {}
{
    ( LOOKAHEAD(2) <DECL>() {} )*
}

void <DECL>() : {}
{
    LOOKAHEAD(3)
    "int" <IDENTLIST>() ";"
    | "int" <IDENT>() "(" <FORMALLIST>() ")"
      <BLOCK>()
}
```

記号表（拡張）

変数が大域（レベル０）で宣言されたもの、関数内（レベル１）で宣言されたものの区別をする必要がある。
まず、エントリの中に level というフィールドを用意する。大域変数の宣言の場合は０、局所（関数内）での変数の場合は１以上（今回の場合は１のみ）に設定するようにする。また、名前が変数、配列、関数のどれに利用されるかを区別する type フィールドを用意する。

```
class Name {
  String ident;
  int type;
  int address;
  int level;  // 今回は0か1
}
```

- type には，0 または 1 または 2 を保持して，変数(0)と配列(1)と関数(2)の区別に利用する． （本来は、このようにマジックナンバーを使うのは良くないので、Enum を使うあるいは、定数変数を用いるべきである。）

表（テーブル）は、レベルごとに別に用意する方法と、別には用意しない方法がある。ここでは別に用意する場合について説明する。まず、コンパイルの開始時には大域用(レベル 0)にテーブルを用意する。関数をコンパイルする場合は、その都度新しいテーブル(レベル 1)を用意する。レベル 1 のテーブルからは、レベル 0 へのテーブルを参照できるようにする。

```
class NameTable{
      Name[] nameTable;
      ....
      int level;          // 記号表のレベル。
      NameTable parent;  // 親テーブル。レベル0の記号表の場合はnull

 NameTable(int max, int lev, NameTable p){
        // 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。
        // レベルを lev とする。
        // 親テーブルを指定する。レベル0 の場合はnull
  }
  ...
 Name searchName(String ident, int type) 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。
 // 自分のレベルに名前が登録されていない場合は、親テーブルを再帰的に探
 // 索する。エラーがあったら例外を投げる
 ...
}
```

変数の参照の例:

```
FACTOR
 ...
    <IDENT>
    {Name entry = nt.searchName(id, 0);  // 0は変数
     (LDV, level - entry.level, entry.address)を生成（配列に対応する場合はLDA を使う）
    }
```

main 関数の定義

関数をコンパイルする際にはBLOCK を処理する前に、レベル 1 の記号表を用意する。メイン関数の場合は、例えば次のようになるだろう。

```
  <INT> <MAIN> "(" ")" {level++; nt = new NameTable(512, level, nt)} BLOCK
```

nt はネームテーブル,level は、現在のレベルである。今回の場合 level++は、level=1 でも良い。

ベースアドレスから 3 個分の記憶域は、システム用に使われるため、局所変数の（相対）アドレスは 3 から始まることに注意する。return(式)で返り値を返すようにする。

```
BLOCK
   {局所変数を３番地から登録するように準備する。}
```

```
    "{" INTDECLS()
   { PUSH 0, n のコードを生成}
       STATEMENTLIST()
    "}"
   { nt = nt.parent; level--; }  // ここでレベル１の記号表を開放（リセット）する。
```

main 関数の呼び出し

main 以外に関数定義がない場合は、main 関数の呼び出しは次のようになる。他に関数定義がある場合、main 関数の開始番地は４ではなくなるので、命令１の CALL０４における飛び先はバックパッチで処理する必要がある。

```
0: PUSH 0 n (n は大域変数の数に応じて)
1: CALL 0 4
2: POP  0 n+1 (+1 は返り値の分)
3: HLT  0 0
4: PUSH 0 m  main 関数のコード(局所変数の数+3)
....
 : EF  0 0  返り値を１つスタックに積んでおく。
```

引数なしの関数宣言と呼び出し

次のようなプログラムがコンパイル、実行できることを目指す。

```
(test-a3.h)
int g;
int test(){
  int a;
  a = 10;
  g = g + a;
  return a;
}

int main(){
  g = 1;
  putint(test());
  putint(g);
  return 0;
}
```

宣言

関数 f を宣言するときは、関数 f はレベル０の記号表に登録する。address には、関数 f の開始アドレスを登録する。f をコンパイルする際には、main と同様、レベル１の記号表を用意する。

```
FUNCDECL
  <INT> <IDENT>
        {関数<IDENT>とその開始アドレスをレベル０の表に登録}
  "(" ")"
             {level++; nt = new NameTable(512, level, nt)}
   BLOCK()
```

呼び出し

文法に関しては、通常の変数参照と関数呼び出しの区別は<IDENT>の次が'(' かどうかで決まるので左くくりだしか、LOOKAHEAD(2)による先読み指定が必要。関数のエントリには開始アドレスが登録されているので、そのアドレスを CALL すればよい。p オペランドは今回は常に１であるが、将来関数の中でネストした関数を使えるようにした場合のため

に、下記のようにレベル差を指定する。

```
FACTOR
 ...
    <IDENT> "(" ")"
    {Name entry = nt.searchName(id, 2);  // 2は関数
     CALL, level - entry.level, entry.address を生成
    }
```

B: 引数ありの関数宣言と呼び出し。(引数は int)

次のようなプログラムがコンパイル、実行できることを目指す。

```
(test-b1.h)
int g;
int test(i){
  int a;
  int b;
  a = 100;
  return a+i;
}
int main(){
  g = test(10);
  putint(g);
  return 0;
}

(test-b2.h)
int g;
int test(i, j){
  int a;
  a = 100;
  return a+i-j;
}
int main(){
  g = test(10,20);
  putint(g);
  return 0;
}
```

記号表の変更点

記号表には関数の引数の数を保存しておく。大域の記号表の場合は0にしておけばよい。

```
class NameTable{

      Name[] nameTable;
      ....
      int argc;          // 関数スコープ（レベル0以外）の場合の引数の数
      int level;         // 記号表のレベル。
      NameTable parent;  // 親テーブル。レベル0の記号表の場合はnull
 ...
}
```

エントリには、関数名の場合には引数の個数を残しておけるようにする。また、変数名の場合、それが仮引数として登録されたものかどうかのフラグを残せるようにする。

```
class Name {
  String ident;
  int address;
  int type;
  int size;
  int level;  // 今回は０か１
  int argc; // 関数の場合、引数の個数。(関数呼び出しの際の引数の数をチェックするため)
  boolean isArg;  // 仮引数かどうか
}
```

- address には，base からの相対アドレスを保持する．配列の場合は，第 0 要素の相対アドレスとする．
- size には，変数や配列が必要とする段数を保持しておく．変数の場合は 1 であるが，配列の場合，配列宣言文の <NUMBER>に対応する数になる．

関数の定義

関数の定義の際、関数名自体（名前、開始アドレス等）はレベル０に、引数および局所変数の名前はレベル１に登録しなければならない。また、引数がある場合には、引数名に対するアドレスはベースからの相対値ではマイナスになるので（引数が a,b,c の３つの場合、それぞれ-3, -2, -1 が相対アドレスとなる)。引数を全部登録した時点で、アドレスを調整するとよい。

```
FUNCDECL
  <INT> <IDENT>
        {関数<IDENT>とその開始アドレスをレベル０の表に登録。
         level++; nt = new NameTable(512, level, nt)}
  "(" FORMALLIST() ")"
        {引数の個数を関数<IDENT>のエントリ(argc)に登録、引数の個数はレベル１の記号表にも登録。(nt.argc= …)
         引数のアドレスをマイナス値になるように調整する（先に調整しておく場合）}
   BLOCK()

  return では、引数の個数を指定する必要がある。
STATMENT ::
  'return' EXPRESSION
      { EF 0, nt.argc のコードを生成}
```

C: 引数ありの関数宣言と呼び出し。(引数は int および配列アドレス)

次のようなプログラムがコンパイル、実行できることを目指す。

```
(test-c1.h)
int g;
int test(i[2]){
  return i[0]+i[1];
}
int main(){
  int b;
  int a[2];
  a[0]=10;
  a[1]=20;
  g = test(a);
  putint(g);
  return 0;
```

```
}
```

パラメータの参照渡しについて(余力のあるもの向け)

```
int a[20];
int func(i[20]){
...
}
```

という宣言があった場合、f の呼び出しは、f(配列名)という呼び出しになることに注意。上記の場合なら例えば、x := func(a);のように呼び出す。すなわち、配列名を単独で参照してよいことになる。

なお、下記のように

```
int func(i[20]){
  int j[10];
  ...
  x := i[10];
}
```

引数の配列 i を参照する場合、i のアドレスを LDA でロードするのではなく、i に「渡される」アドレスを LDV でロードする必要がある。

局所の j[10]を参照する場合、

```
..
LDA 0, 0   (j のアドレス)
LDC 0, 10
AD 0, 0
LDI 0, 0 (代入なら STI)
..
```

であるが、引数で与えられた i[10]を参照する場合、

```
..
LDV 0, -1   (-1 番地(第一引数のアドレス)にはアクセスしたい配列のアドレスが入っている)
LDC 0, 10
AD 0, 0
LDI 0, 0 (代入なら STI)
..
```

となることに留意する必要がある。