6. インタープリタ

♣ これまでに作成した抽象構文木のデータ構造を用いて, Mini-C 言語のインタープリタを作成する.

♣ C++ によるオブジェクト指向的なプログラミングを体験するのも目的の一つである.

☆ 課題の前半は「6.2.11 関数呼び出し文」の 課題 6.11 まで, 後半はそれ以降

6.1 インタープリタ実装の考え方

前回の演習で, 一つの構文要素に対して抽象構文木の一つのクラスを定義した. それぞれのクラスに対して, そのクラスの情報を表示する `print` メソッドを定義することにより, プログラム全体を表示することができた. これと同じように, 抽象構文木の各クラスに対して, それを実行したときの動作を実現するメソッド (`run` とする) を実装することにより, プログラム全体のインタープリタを作ることができる.

もう少し具体的に見てみると, 次のようになる.

- 「式」の実行結果は, その式の値を計算して得られる値 (`int` 型) と考える. 例えば,
  - 整数定数 `7` に対する実行結果は, 整数値 `7` である.
  - 変数 `x` に対する実行結果は, 計算のその時点での `x` の値である. 各変数の値を

`"a"`	`3`
`"b"`	`4`
`"x"`	`10`

    のような表 (以後, 「変数表」と呼ぶことにする) で記憶しておき, その時点での `x` の値を読み出して返せばよい. 変数表は STL の `map` で実装できる.
  - 二項演算　式$_1$ `+` 式$_2$　の実行結果は, この式を計算した結果である. 式$_1$ と 式$_2$ をそれぞれ実行し, 得られた値の和を返せばよい.
- 「文」の実行は, それぞれの文の意味に応じたものになる.
  - 代入文　`x =` 式`;`　の実行は,
    1. まず右辺の 式 を実行してその値を求め,
    2. 変数表の `x` の値を求めた値に書き換える

    ことにより行える.
  - 文のリスト　`{` 文$_1$`;` 文$_2$`;` $\cdots$`;` 文$_m$ `}`　の実行は, それぞれの文をこの順に実行することである.
  - `if (`式`) {` 文$_1$`} else {` 文$_2$`}`　の実行は,
    1. まず, 式 を実行して値を求め,
    2. その値が `0` 以外ならば, 文$_1$ を実行し,
    3. その値が `0` ならば, 文$_2$ を実行する

    ことにより行える.
- 「関数」の実行は,
  - 受け取った引数とローカル変数を変数表に書き込み,
  - 本体を実行する

  ことにより行える.
- 与えられたプログラム全体の実行は, `main` 関数を実行することにより行える.

本演習では, このような考え方に基づき, Mini-C 言語のインタープリタを作成する.

6.2 インタープリタ実装の演習

6.2.1 定数式 (Exp_constant)

課題 6.1 7 や 'x' 等の「定数式」の実行メソッド Exp_constant::run を作成せよ.

1. run は, 全ての「式」に共通な仮想関数として実装する. このため, まず抽象基底クラス Expression のメソッドとして run を追加する.

```
virtual int run(
    std::map<std::string,Function*>& func,   // (1) 関数表 //
    std::map<std::string,int>& gvar,         // (2) グローバル変数表 //
    std::map<std::string,int>& lvar          // (3) ローカル変数表 //
) const = 0;
```

   – 作業的には, ast.h の Expression の class 宣言 (の virtual void print ··· の後に) この宣言を追加する.

   宣言にあるように, 式の run は 3 つの表を参考しながら式の値を計算し, 得られる結果 (整数値) を返す

   (1) 関数表は, 関数名とその実体 (その関数の抽象構文木へのポインタ) の対応を記憶する表である.

   (2) グローバル変数表は, グローバル変数の値を記憶する表である (変数名をキーにして値を得る).

   (3) ローカル変数表は, ローカル変数の値を記憶する表である (変数名をキーにして値を得る).

2. 次に, Exp_constant の run の定義と実装を与える. 定数式の値とは, 保持している定数値 (value()[1]) なので, これを返すように書けばよい. 短いので, ast.h のクラス宣言の中 (void print ··· の後あたり) に実装まで書いてしまう.

```
int run(
  std::map<std::string,Function*>& func,
  std::map<std::string,int>& gvar,
  std::map<std::string,int>& lvar
) const { return value(); }
```

3. map を使うために, ast.h の冒頭で <map> をインクルードせよ. さらに, std::map<std::string,Function*> の宣言が class Function の宣言よりも前にあると, コンパイラには Function* が何の型なのか判らないので, #include<map> の次の行ぐらいに

```
class Function;
```

   と宣言しておく.

4. これはコンパイルの都合だが, この時点で Exp_function にも run の定義と仮の実装をしておかなければならない[2]. ast.h の class Exp_function の中に

```
int run(
  std::map<std::string,Function*>& func,
  std::map<std::string,int>& gvar,
  std::map<std::string,int>& lvar
) const;
```

[1] value_ でもよい.

[2] St_function のメンバーに Exp_function を使っている関係で, Exp_function にも仮想関数 run の宣言と実装を与えておかないと, コンパイルエラーが出る.

と宣言し, ast.cpp の中に,

```
int Exp_function::run(
  std::map<std::string,Function*>& func,
  std::map<std::string,int>& gvar,
  std::map<std::string,int>& lvar
) const
{
  return 0;  // 仮なので, 何でもよい.
}
```

と仮の実装を与えよ. (本物の実装は 課題 6.5 で作成する.)

5. 次のようなプログラムで, 動作を確認せよ.

```
// 課題 5.1 と同じ抽象構文木
Expression* c1 = new Exp_constant(Type_INT, 7);
Expression* c2 = new Exp_constant(Type_CHAR, 'x');
// 実行とその結果の表示
std::map<std::string,Function*> func;
std::map<std::string,int> gvar;
std::map<std::string,int> lvar;
std::cout << c1->run(func, gvar, lvar); std::cout<<std::endl;
std::cout << c2->run(func, gvar, lvar); std::cout<<std::endl;
```

実行して

```
7
120   … 'x' のコード
```

が得られれば OK.

☆ プログラムは特に添付しなくてよい. 演習の経過と感想のみ記せ (以下, 最後の演習まで同様).

6.2.2 変数式 (Exp_variable)

課題 6.2 a や x 等, 式中に現れる変数の実行メソッド Exp_variable::run を作成せよ.

1. ast.h の class Exp_variable に, run の宣言を追加せよ.
2. ast.cpp 中に Exp_variable::run の実装を作れ.

   変数の値は, 次の手順で求める.

   (1) まずローカル変数表をみて, その変数があればその値を返し,

   (2) なければグローバル変数表をみて, その変数があればその値を返し,

   (3) そこにもなければ,「そのような変数は宣言されていない」とエラーメッセージを出して, 強制終了する.

   参考までに, (1) は次のように書ける.

```
std::map<std::string,int>::const_iterator p;
if ((p=lvar.find(name()))!=lvar.end()) {
   return p->second;
}
```

3. 次のプログラムでテストせよ.

```
Expression* v_a = new Exp_variable("a");
Expression* v_n = new Exp_variable("n");
Expression* v_x = new Exp_variable("x");
Expression* v_p = new Exp_variable("p");
// 3 つの表の宣言
std::map<std::string,Function*> func;
std::map<std::string,int> gvar;
std::map<std::string,int> lvar;
// 変数の値を設定
lvar["a"] = 4;
lvar["x"] = 2000;
gvar["x"] = -95;
gvar["n"] = 10;
// 実行と表示
std::cout << v_a->run(func,gvar,lvar); std::cout<<std::endl;
std::cout << v_n->run(func,gvar,lvar); std::cout<<std::endl;
std::cout << v_x->run(func,gvar,lvar); std::cout<<std::endl;
std::cout << v_p->run(func,gvar,lvar); std::cout<<std::endl;
```

結果,

```
4                       … a (ローカル変数)
10                      … n (グローバル変数)
2000                    … x (ローカル変数の方)
undefined variable p    … 定義されていない
```

と表示されれて終了すれば OK.

6.2.3 単項演算式 (Exp_operation1)

課題 6.3 +式 や -式 等の「単項演算式」の実行メソッド Exp_operation1::run を作成せよ.

1. 単項演算式の計算は, まずオペランド (operand()) の run メソッドを実行してその値を求め, それに対して演算子 (operation()) に対応した演算を行い, 得られた値を返せばよい. 例えば, オペランドの値が 3 になり, 演算子が Operator_MINUS であれば, -3 を返せばよい.

   ☆ 安全のため, 最初に operand() が NULL でないかどうかチェックする (「オペランドが定義されていない」というエラーにするか, 面倒なら単に assert だけでもよい) ことを推奨する.

2. 次のようなプログラムで動作を確認せよ.

```
// 5.03 そのまま
Expression* v = new Exp_variable("n");
Expression* o = new Exp_operation1(Operator_MINUS, v);
// テスト
std::map<std::string,Function*> func;
std::map<std::string,int> gvar;
std::map<std::string,int> lvar;
lvar["n"] = 31;
std::cout << o->run(func, gvar, lvar); std::cout<<std::endl;
```

n=31 に対して -n を計算しているので, -31 が表示されれば OK.

6.2.4 二項演算式 (Exp_operation2)

課題 6.4 式$_1$ + 式$_2$ のような「二項演算式」の実行メソッド Exp_operation2::run を作成せよ.

1. 考え方は単項演算式と同様である.
2. 次のようなプログラムで動作を確認せよ.

```
// 5.04 そのまま
Expression* v = new Exp_variable("n");
Expression* c1 = new Exp_constant(Type_INT, 1);
Expression* o1 = new Exp_operation2(Operator_PLUS, v, c1);
Expression* c2 = new Exp_constant(Type_INT, 2);
Expression* o2 = new Exp_operation2(Operator_MUL, o1, c2);
// テスト
std::map<std::string,Function*> func;
std::map<std::string,int> gvar;
std::map<std::string,int> lvar;
lvar["n"] = 31;
std::cout << o1->run(func, gvar, lvar); std::cout<<std::endl;
std::cout << o2->run(func, gvar, lvar); std::cout<<std::endl;
```

n=31 に対し, o1=n+1 と o2=o1*2 を計算して出力しているので, 32 と 64 が表示されれば OK.

6.2.5 関数式 (Exp_function)

課題 6.5 gcd(a,b+5) や getint() 等の関数呼出しを表す「関数式」の実行メソッド Exp_function::run を作成せよ.

1. ast.cpp 中に Exp_function::run を実装する.

   「関数式」の実行の流れは次のようになる.

```
(1) 引数の式のリストから, その値のリストを求める;
(2) if (関数名が "getint") { getint の処理をする; }
(3) else if (関数名が "getchar") { getchar の処理をする; }
(4) else if (関数名が "putint") { putint の処理をする; }
(5) else if (関数名が "putchar") { putchar の処理をする; }
(6) else { 一般の関数の処理をする; }
```

   (1) 引数の値リストの計算

   aaa(a+1, b*2, 5) という関数呼出しの場合, 引数リストは

   (a+1, b*2, 5)

   である. 今, a=3, b=5 であれば, 引数の各式の値を計算して,

   (4, 10, 5)

   という整数のリストを作成する. 引数 (式) のリストは args() に格納されている. この要素を順にたどりながら各々を run で実行し, 得られた結果を順に整数のリスト std::list<int> i_args に格納せよ.

   (2) getint 関数

   一般の関数の処理に先立ち, まず組込み関数を処理する. 関数名 (name()) が "getint" であれば, 整数を一つ std::cin から入力し, その値を返す.

```
int i;
std::cin >> i;
return i;
```

(2) getchar 関数

前項の getint と同じ. 読み込みの際, 整数変数ではなく文字変数に読み込めばよい.

(3) putint 関数

putint は値を返さない関数なので, 本来 Exp_function では扱わないのだが, 今回の St_function 実装が Exp_function を呼び出す形になっているので, この部分で作成する. 処理としては, 引数値のリスト i_args の 1 番目を std::cout に出力するだけである. 返り値は使用しないので何でもいい.

```
int i = i_args.front();  // 引数値のリストの先頭
std::cout << i;
return 0;
```

(4) putchar 関数

putint と同様.

(5) 一般の関数

関数表 func を検索し, この関数名に対応する関数のポインタ f を求める (なければ「そのような関数が定義されていない」というエラー). そして, 関数 f の run メソッドを呼び出す. run の呼び出しの引数には, 関数表 func, グローバル変数表 gvar, および引数の値リスト i_args を渡す.

```
std::map<std::string,Function*>::const_iterator p;

...
else {
  if ((p=func.find(name()))!=func.end()) {
    Function* f = p->second;
    return f->run(func,gvar,i_args);
  }
  else {
    エラー
  }
}
```

Function::run は 課題 6.12 で実装するが, この時点で何か実装を与えておかなければ動かないので, ast.h の class Function の中に,

```
int run(
  std::map<std::string,Function*>& func,
  std::map<std::string,int>& gvar,
  std::list<int>& i_args
) const;
```

を追加し, ast.cpp には「1 番目の引数をそのまま返す」という仮の実装を作っておく.

```
int Function::run(
  std::map<std::string,Function*>& func,
  std::map<std::string,int>& gvar,
  std::list<int>& i_args
) const
{
  return i_args.front();   // 仮の実装
}
```

2. ホームページより 6_05.cpp をダウンロードし, コンパイル&実行せよ. このプログラムは,

```
getint();
getchar();
putint(5);
putchar('U');
asum(n);
```

の各々抽象構文木を作り, n = 7 を初期値としてこれらの抽象構文木を順に実行するものである. 正しい結果が得られることを確認せよ.

6.2.6 代入文 (St_assgin)

これより, 「文」に移る.

課題 6.6 「代入文」の実行メソッド St_assgin::run を作成せよ.

1. run は, 全ての「文」に共通な仮想関数として実装する. まず, 抽象基底クラス Statement に run を宣言する.

```
virtual Return_t run(
  std::map<std::string,Function*>& func,
  std::map<std::string,int>& gvar,
  std::map<std::string,int>& lvar
) const = 0;
```

文の run は, 式と同様, 関数表 func, グローバル変数表 gvar, ローカル変数表 lvar を受け取って文の実行結果を計算する.

文の run の返り値であるが, 一般に a=b+3; 等の普通の文に関しては特に返り値は必要無いが, return 文では返り値が生じる. また,

```
while(i<n) {
  if (n%r==0) return r;
  i++;
}
```

のように, return 文を含む複合文の場合には, return 文が実行されたかどうかにより, 計算を継続するか終了するかが変わってくる.

このように, 「文」の run では, return 文が返す値も必要だが, その文の中で return 文が実行されたかどうかという情報も必要である.

そこで, 今回の実装では, 文の run の返り値を単なる int 型ではなく, 2 つの情報を同時に保持する Return_t 型にする.

その定義は, 次の通りである. (ast.h の冒頭の #include の後あたりに書き込め).

```
struct Return_t
{
  bool val_is_returned;  // return 文が実行されたかどうか (true/false)
  int return_val;        // return 文が実行された場合, その返り値
  Return_t() :  val_is_returned(false), return_val(0) {}
  Return_t(bool r, int v) :  val_is_returned(r), return_val(v) {}
};
```

☆ C++ では, struct にも class と同じく名前をつけたり, コンストラクタやデストラクタ, メソッドを定義したりでき, 機能的にほとんど同じである. private メンバがあって, メソッドを通じてのみメンバのデータを操作許す場合には class を, 全メンバが public で, これを外部から直接操作するのを許す場合には struct を用いる, という使い分けをすることが多い.

代入文等のように内部で return 文が実行されない場合には Return_t(false,0) を返し, 文中で return 文が実行されてその返り値が例えば 7 であった場合には Return_t(true,7) を返す.

文からの返り値が Return_t rd である場合, その文の中で return が実行されたかどうかは, rd.val_is_returned が true かどうかで判る. 実行された場合の返り値は rd.return_val で参照できる.

2. ast.h のクラス St_assign の宣言中に run の宣言を追加せよ. Statement と同様である[3].
3. ast.cpp 中に St_assign::run を実装せよ.
   (1) 念のため, 冒頭で左辺式と右辺式が NULL でないことを確認しておくことを推奨する.

   ```
   assert(lhs());
   assert(rhs());
   ```

   だけでもよい.
   (2) 右辺式 (rhs()) の run を起動して, その値を求める (i_rhs とする).
   (3) 左辺 (lhs()) の変数をローカル変数表とグローバル変数表の中に探し, その値を i_rhs に書き換える. Exp_variable と同じような制御で, まずローカル変数表を検索し, なければ次にグローバル変数表を検索する. いずれにもその変数が登録されていなければ, エラーとする.
   (4) 一番最後に, return 文が実行されていない旨を返り値で伝える.

   ```
   return Return_t(false,0);
   ```

4. 次のプログラムで動作確認せよ.

```
// i = 0; という代入文 (5.06 のまま)
Exp_variable* v1 = new Exp_variable("i");
Expression* c1 = new Exp_constant(Type_INT, 0);
Statement* s1 = new St_assign(v1,c1);
// テスト
std::map<std::string,Function*> func;
std::map<std::string,int> gvar;
std::map<std::string,int> lvar;
std::cout << "before" <<std::endl;
// i の初期値を 5 に設定して実行
lvar["i"] = 5;
Return_t rd = s1->run(func, gvar, lvar);
std::cout << "i = " << lvar["i"] <<std::endl;
```

i = 5 を初期値として代入文 i = 0; を実行しているので, 結果 “i = 0” が表示されれば OK.

[3]末尾の = 0 は取るように. 念のため.

6.2.7 文のリスト (St_list)

課題 6.7 「文のリスト」の実行メソッド St_list::run を作成せよ.

1. 基本的にリスト中の文を順番に実行するだけなので, iterator で要素を順にたどりながら各要素の run を起動すればよい.

   ただし, 途中で return 文が実行された場合は, そこで実行を打ち切って返り値を返さなければならない. 逆に, return 文が一度も実行されなかった場合は, 最後に Return_t(false,0) を返さなければならない.

```
for (std::list<Statement*>::const_iterator it = statements().begin();
      it != statements().end(); it++) {
  assert(*it);  // 念のため NULL でないことを確認
  Return_t rd = (*it)->run(func, gvar, lvar);  // 実行
  if (rd.val_is_returned) return rd;  // return 文が実行されていたら, rd をそのまま返す
}
return Return_t(false,0);  // return 文が一度も実行されていない場合
```

2. 次のプログラムで動作を確認せよ.

```
// 5.07 のまま
// s = 0;
Exp_variable* v1 = new Exp_variable("s");
Expression* c1 = new Exp_constant(Type_INT, 0);
Statement* s1 = new St_assign(v1,c1);
// i = -n;
Exp_variable* v2 = new Exp_variable("i");
Exp_variable* v3 = new Exp_variable("n");
Expression* e1 = new Exp_operation1(Operator_MINUS, v3);
Statement* s2 = new St_assign(v2,e1);
std::list<Statement*> slist;
slist.push_back(s1);
slist.push_back(s2);
Statement* s3 = new St_list(slist);
// テスト
std::map<std::string,Function*> func;
std::map<std::string,int> lvar;
std::map<std::string,int> gvar;
lvar["s"] = 123;
lvar["i"] = 456;
lvar["n"] = 5;
Return_t rd = s3->run(func, gvar, lvar);
std::cout << "s = " << lvar["s"] <<std::endl;
std::cout << "i = " << lvar["i"] <<std::endl;
std::cout << "n = " << lvar["n"] <<std::endl;
```

   このプログラムは, s=123, i=456, n=5 を初期値として,

```
s = 0;
i = -n;
```

   を実行するので, 次のような表示が得られれば OK.

```
s = 0
i = -5
n = 5
```

6.2.8 if 文 (St_if)

課題 6.8 「if 文」の実行メソッド St_if::run を作成せよ.

1. まず, 条件部 condition() を実行して値を求め, 真 (0 以外) なら then_part() を, そうでなければ else_part() を実行すればよい.

   ただし, 条件部が偽でも else_part() が存在しなければ実行しない. (同様に, 条件部が真でも then_part() が存在しなければ実行しない.)[4]

   また, then_part() の run の返り値 rd の rd.val_is_returned が true なら then_part() の中で return 文が実行されたので, その場で rd を返す. else_part() についても同様. 一番最後まで来た場合には, Return_t(false,0) を返す. (前課題の St_list とよく似ているので, それを参考にせよ.)

2. 次のプログラムで動作確認せよ.

```
Statement* s = make_if(); // 前回の演習で作成したもの
std::map<std::string,Function*> func;
std::map<std::string,int> gvar;
std::map<std::string,int> lvar;
// 真の場合
lvar["i"] = -5;
lvar["s"] = 10;
Return_t rd1 = s->run(func, gvar, lvar);
std::cout << "s = " << lvar["s"] <<std::endl;
// 偽の場合
lvar["i"] = 7;
lvar["s"] = 10;
Return_t rd0 = s->run(func, gvar, lvar);
std::cout << "s = " << lvar["s"] <<std::endl;
```

   これは,

```
if (i<0) s = s - i;
else     s = s + i;
```

   というプログラムに

   s=10, i=-5

   s=10, i=7

   を与えてそれぞれ実行した結果なので, 次のような出力が得られれば OK.

```
s = 15
s = 17
```

6.2.9 while 文 (St_while)

課題 6.9 「while 文」の実行メソッド St_while::run を作成せよ.

[4] condition() が NULL ならエラーにすべきだが, then_part() や else_part() は NULL でもエラーにしてはならない.

1. 考え方は if 文と同様である.
2. 次のようなプログラムで動作を確認せよ.

```
Statement* s = make_while();  // 前回の演習で作成したもの

std::map<std::string,Function*> func;
std::map<std::string,int> gvar;
std::map<std::string,int> lvar;
lvar["i"] = -3;
lvar["n"] =  3;
lvar["s"] =  0;
Return_t rd = s->run(func, gvar, lvar);
std::cout << "i = " << lvar["i"] <<std::endl;
std::cout << "n = " << lvar["n"] <<std::endl;
std::cout << "s = " << lvar["s"] <<std::endl;
```

これは

```
while(i<=n) {
  if (i<0) s = s - i;
  else     s = s + i;
  i = i + 1;
}
```

に i=-3, n=3, s=0 を与えて実行した結果なので, 次のような出力が得られれば OK.

```
i = 4
n = 3
s = 12
```

6.2.10 return 文 (St_return)

課題 6.10 「return 文」の実行メソッド St_return::run を作成せよ.

1. 本体は次の通り.

```
assert(value());
int rv = value()->run(func,gvar,lvar);
return Return_t(true,rv);
```

2. ホームページより次の 6_10.cpp をダウンロードし, コンパイル&実行せよ. このプログラムは,

```
while(i<=n) {
  if (i<0) s = s - i;
  else     return s;
  i = i + 1;
}
```

というものであり, 前課題のプログラムにおいて i が 0 になった時点で計算を打ち切る. したがって, 実行結果は次のようになれば OK.

```
i = -3
n = 3
s = 0
rd.val_is_returned = 1
rd.return_val = 6
```

6.2.11 関数呼び出し文 (St_function)

課題 6.11 「関数呼び出し文」の実行メソッド St_function::run を作成せよ.

1. function_ の run を呼び出して実行し, Return_t(false,0) を返す.
2. 次のようなプログラムで動作を確認せよ.

```
std::list<Expression*> arglist;
arglist.push_back(new Exp_variable("a"));
Statement* s = new St_function("putint", arglist);

std::map<std::string,Function*> func;
std::map<std::string,int> gvar;
std::map<std::string,int> lvar;
lvar["a"] = -3249;
Return_t rd = s->run(func, gvar, lvar);
```

a=-3249 のときに putint(a) を実行しているので, -3249 が出力されれば OK.

☆☆☆ 課題の前半はここまで ☆☆☆

6.2.12 関数 (Function)

「関数」の実行メソッド Function::run は,

- 関数表 func
- グローバル変数表 gvar
- 引数の値のリスト i_args (std::list<int>&)

を受け取り,

- ローカル変数表 lvar を作り,
- 関数の本体 body(func,gvar,lvar) を実行し,
- 返り値 (int 型) を返す,

というものである.

課題 6.12 Function::run を完成させよ.

☆ 以前に作成した仮の実装 return i_args.front(); は消去すること.

1. ローカル変数表 lvar を宣言する.

```
std::map<std::string,int> lvar;
```

2. 引数の名前と値をローカル変数表に登録する.

   引数のリストが (a,b,c) で, 引数の値のリストが (10,7,23) であれば,

```
lvar["a"] = 10;
lvar["b"] = 7;
lvar["c"] = 23;
```

   のように, 名前と値の組を lvar に書き込む. 引数の名前は変数のリスト args() から, 値は整数値のリスト i_args から取得できるので, リストの各要素についてこの操作を行う.

3. ローカル変数をローカル変数表に登録する.

   ローカル変数が x, y, z であれば,

```
lvar["x"] = 0;
lvar["y"] = 0;
lvar["z"] = 0;
```

   または,

```
lvar["x"];
lvar["y"];
lvar["z"];
```

   として lvar にローカル変数を登録する.

4. 本体を呼び出す.

```
Return_t rd = body()->run(func, gvar, lvar);
```

5. 本体の返り値を返す.

   整数値だけを返すので,

```
return rd.return_val;
```

6. 次のプログラムで動作確認せよ.

```
Function* function_asum = make_function_asum();  // 前回の演習で作成したもの

std::map<std::string,Function*> func;
std::map<std::string,int> gvar;
std::list<int> i_arglist;
i_arglist.push_back(3);
int r = function_asum->run(func, gvar, i_arglist);
std::cout << r << std::endl;
```

   これは

```
int asum(int n)
{
  int s; int i;
  s = 0;
  i = -n;
  while(i<=n) {
    if (i<0) s = s - i;
    else     s = s + i;
    i = i + 1;
  }
  return s;
}
```

という関数に n = 3 を与えて実行し, その返り値を表示させているので, 12 が表示されれば OK.

7. 次のプログラムを実行し, 関数呼び出しが正しく実行できることを確認せよ.

```
std::list<Expression*> arglist;
arglist.push_back(new Exp_constant(Type_INT, 5));
Expression* exp_function_asum = new Exp_function("asum", arglist);

std::map<std::string,Function*> func;
std::map<std::string,int> gvar;
std::map<std::string,int> lvar;
func["asum"] = make_function_asum();
int asum5 = exp_function_asum->run(func, gvar, lvar);
std::cout << "asum(5) = " << asum5 <<std::endl;
```

asum(5); という関数呼び出しの計算を行っているので, asum(5) = 30 と表示されれば OK.

6.2.13 プログラム (Program)

いよいよプログラム全体の実行メソッド Program::run, すなわちインタープリタの本体を完成させる. Program::run は,

- グローバル変数表 gvar を作り,
- 関数表 func を作り,
- 関数 main を実行し, 返り値を返す.

というものである.

課題 6.13 Program::run を完成させよ.

1. グローバル変数表 gvar を宣言し, varlist() の変数を gvar に登録する.

   ☆ 前演習の Function::run のローカル変数の登録と同様である.

2. 関数表 func を宣言し, funclist() の関数を func に登録する.
3. main を呼び出す.

```
std::list<int> iargs;
int v = main()->run(func, gvar, iargs);
return v;
```

4. 次のプログラムで動作を確認せよ.

```
// プログラムの抽象構文木
std::list<Variable*> gvar;
std::list<Function*> func;
gvar.push_back(new Variable(Type_INT, "g"));
func.push_back(make_function_asum());
Program* p = new Program(gvar, func, make_function_main());
// 実行
p->run();
```

ただし, make_function_main は, 次の main 関数を抽象構文木を構成する関数である.

```
int main()
{
  int a;
  g = 3;
  a = asum(g);
  putint(a);
}
```

これを実行すると, 12 が表示されるはず.

課題 6.14 前回の課題で作成した factor.cpp の

```
prog->print(std::cout);
```

をコメントアウトし, 代わりに

```
prog->run();
```

を加えてコンパイル&実行し, 下記のように素因数分解が行われることを確認せよ.

```
n=360
2*2*2*3*3*5
```

☆ 完成した ast.h と ast.cpp をレポートに添付せよ.

コメント

これまでの演習を通じて感じたかも知れないが, C++ 等によるオブジェクト指向的プログラミングでは, クラスやそのインタフェースがうまく設計できれば, クラス毎のメソッドの実装は比較的単純な作業になり, 大規模なプログラムの開発の見通しが良くなる. ただし, クラスやインタフェースの設計は本質的で, ここにはセンスが必要になる. プログラムの行数が思ったより多くなったり, コンパイルの型チェックが厳密なのは煩わしいが, 多人数で大規模なプログラムを開発する場合には, 型の整合性を取ることが, バグの防止や保守性の向上のために極めて重要になって来る.

Nagisa ISHIURA