5. 抽象構文木

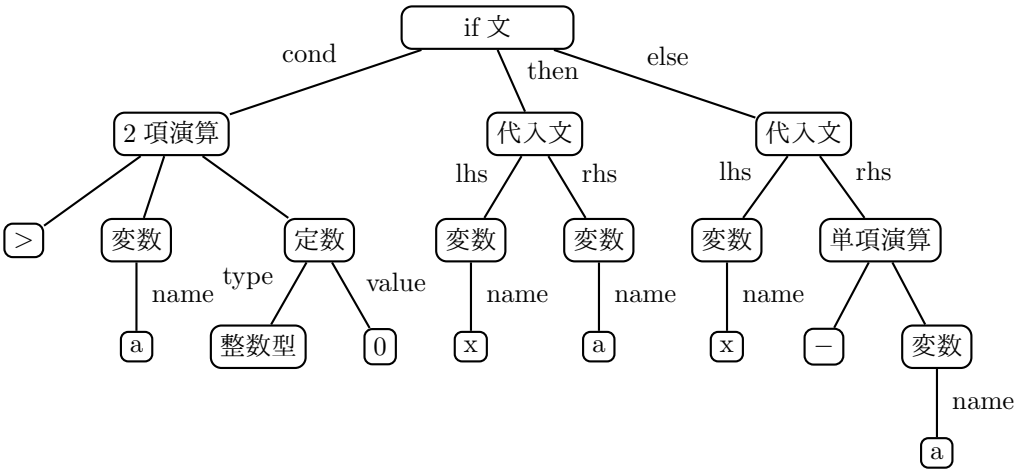
♣ C++ でのデータ構造の構築を行う例題として, プログラムを木構造として表現する抽象構文木を取り上げる. 継承によるポリモーフィズムが有効な例でもある.

☆ 課題の前半は「5.3.9 文の並び」の 課題 5.7 まで, 後半はそれ以降

5.1 抽象構文木とその例

抽象構文木は “Abstract Syntax Tree” の訳で, プログラムの文法構造を木 (tree) の形で表したものである. “AST” と略されることもある. 例えば, 次のようなプログラム (の一部) は, 下のような木 (tree) 構造で表現できる.

```
if ( a > 0 ) {
  x = a;
}
else {
  x = -a;
}
```



この抽象構文木は, この「if 文」が

1) 条件を表す式 (cond)
2) 条件が成立するときに実行する文 (then)
3) 条件が成立しないときに実行する文 (else)

から成ることを表している. それぞれの式や文も木で表される. cond の式は「2 項演算」であり, 「変数」a と整数型定数 0 に対して演算 < を行っている. then 部は「代入文」であり, 左辺 (lhs) の変数 x に右辺 (rhs) の変数 a を代入するものである. else 部も「代入文」である. 代入の右辺は「単項演算」で, 変数 a に対して符号反転 (−) を行うことを表している.

5.2 C++ における抽象構文木の表現

基本的に, 一つの構文要素を表す節点に対して一つのクラスを定義する. 例えば, 上記の「if 文」を表す節点に対して一つのクラス `St_if` を定義する[1]. 同様に,

「2 項演算」に対しては `Exp_operation2` というクラスを[2],

「変数」に対しては `Exp_variable` というクラスを,

「定数」に対しては `Exp_constant` というクラスを,

「代入文」に対しては `St_assign` というクラスを,

というようにそれぞれ定義する.

さて, クラス `St_if` は, 3 つの子へのポインタを格納するメンバー `cond`, `then`, `else` を持つが, ここで例えば `cond` の型は何にしたらよいだろうか?

先程の図の場合には「2 項演算」の節点を指していたので, `Exp_operation2*` とすればいいが, これだと「if 文の条件には 2 項演算しか書けない」ことになってしまう. `if(a) {...}` のように条件部に変数だけが書かれることもあるし, 定数だけ書かれることもあり得る. `St_if` のメンバー `cond` には, `Exp_operation2` だけでなく, `Exp_operation1`, `Exp_variable`, `Exp_constant` 等, 「式」に成り得るあらゆるクラスのインスタンスを指すポインタが格納出来なければならない.

C++ ではこれをポリモーフィズムで解決する[3]. 「式」を表す抽象基底クラスとして `Expression` を定義し, `Exp_operation2`, `Exp_operation1`, `Exp_variable`, `Exp_constant` 等, 式になるものを全てその派生クラスにする. `cond` を `Expression*` 型にすれば, これらの派生クラスのポインタなら何でも格納することができる.

「if 文」の `then` や `else` に関しても同様である. このメンバーには, 「if 文」「代入文」の他「while 文」「return 文」等, 「文」として扱えるあらゆる「文」クラスのインスタンスへのポインタを格納しなければならない. このためには, 「文」の抽象基底クラス `Statement` を定義し, 「文」になる全てのクラスをこのクラスの派生クラスとすればよい.

5.3 抽象構文木の構築の演習

5.3.1 概要

次に BNF を示す Mini-C 言語[4]の抽象構文木の各クラスを定義するのが今回の演習である. クラスが定義できたら, C++ のプログラムによって抽象構文木のインスタンスを構築する. 構築できた抽象構文木の内容を表示するメソッドも実装することによって, 正しく抽象構文木ができていることを確認する. 後の演習では, この抽象構文木を用いたインタープリタを作成する.

Mini-C 言語の BNF (簡易版)

```
1.  プログラム ::= ( 宣言頭部 ( ; | 関数宣言尾部尾部 ) )*
2.  宣言頭部 ::= 型 ID
3.  関数宣言尾部 ::= "(" (ε | 変数宣言 ( "," 変数宣言 )* ) ")" 関数本体
4.  関数本体 ::  = "{" (変数宣言 ";")*  文* "}"
5.  変数宣言 ::= 宣言頭部 変数宣言尾部
6.  型 ::= "int" | "char"
7.  文 ::= ";" | "{" 文* "}" | if 文 | while 文 | return 文 | 関数呼出し ";" | 代入文
8.  if 文 ::= "if" "(" 式 ")" 文 ( ε | "else" 文 )
```

[1] `St_` は Statement の略.

[2] `Exp_` は Expression の略.

[3] ちなみに C では, 共用体 (union) を用いて解決するのが一般的.

[4] 「コンパイラ」の授業で扱う Mini-C 言語よりも少し簡略化してある

```
 9.  while文 ::= "while" "(" 式 ")" 文
10.  return文 ::= "return" "(" 式 ")" ";"
11.  代入文 ::= 変数名 "=" 式  ";"
12.  変数名 ::= ID
13.  式 ::= 式2 ( ( "<" | ">" | "<=" | ">=" | "==" | "!=" ) 式2 )*
14.  式2 ::= ( ε | "+" | "-" ) 式3 ( ( "+" | "-" ) 式3 )*
15.  式3 ::= 式4 ( ( "*" | "/" | "%" ) 式4 )*
16.  式4 ::= 定数 | "(" 式 ")" | 関数呼出し | 変数名
17.  定数 ::= INTERGER | CHAR
17.  関数呼出し ::= 関数名 "(" 引数リスト ")"
18.  関数名 ::= ID
19.  引数リスト ::= ε| 式  ( "," 式 )*
```

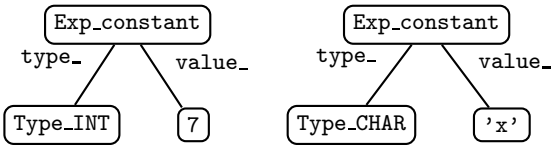
5.3.2 準備

講義のホームページの「プログラム」より, 次のファイルをダウンロードせよ.

- ast.h … 抽象構文木クラスのヘッダファイル (の雛型)
- ast.cpp … 抽象構文木クラスの実装ファイル (の雛型)

5.3.3 定数式 (Exp_constant)

まず, 「定数」のクラスを作る. 下図は, それぞれ整数定数 7 と文字定数 'x' に対する抽象構文木である.



クラス Exp_constant の仕様は次の通りである[5].

- 継承関係

  「式」を表す抽象基底クラス Expression の派生クラスである.

- メンバー

  1. Type type_ … その定数の型を表す (整数型なら Type_INT, 文字型なら Type_CHAR).

     Type は enum 型として次のように定義してある (ast.h の冒頭部).

     ```
     enum Type {
       Type_INT,   // int 型
       Type_CHAR   // char 型
     };
     ```

     ☆ このように書くと, 実際には Type_INT と Type_CHAR はそれぞれ 0, 1 という整数値に変換されるので, 整数型を 0, 文字型を 1 と取り決めたのと同じだが, プログラム中に定数を直書きすることは避けるべきなので, このように enum 型にするのが一般的である.

  2. int value_ … 定数の具体的な値を保持する.

- メソッド

[5] 整数定数のクラスと文字定数のクラスを別に作ってもよいが, あまりクラスを増やしたくないので, ここでは「定数」という一つのクラスにまとめている.

1. Exp_constant(Type t, int i)

   型が t で値が i であるような「定数」の節点 (のインスタンス) を作るコンストラクタ.

2. ~Exp_variable() … デストラクタ. 特に何もしない.
3. int value() const … value_ の値を読み出す.
4. Type type() const … type_ の値を読み出す.
5. void print(std::ostream& os) const … この節点の内容 (すなわち定数の値) を表示する.

まず「定数」クラスを定義する前に, その基底となる「式」のクラス Expression を定義する. 次のリストのようになり, メンバーは特に持たない.

```
 1:  class Expression
 2:  {
 3:  public:
 4:     Expression() {}   // コンストラクタ
 5:     virtual ~Expression() {}  // デストラクタ
 6:     virtual void print(std::ostream& os) const = 0;  // 表示メソッド
 7:  private:
 8:     Expression(const Expression&);  // コピーコンスタラクタは禁止
 9:     Expression& operator=(const Expression&);  // 代入演算は禁止
10:  };
```

- print は, でき上がった抽象構文木を表示するためのメソッドである. 継承クラスに共通のインタフェースで定義されるので, 仮想関数にする.
- コピーコンストラクタと代入演算は, 必要が無ければ禁止しておく.

「定数」を表すクラス Exp_constant は, Expression の派生クラスとして次のように定義できる.

```
 1:  class Exp_constant : public Expression
 2:  {
 3:  private:
 4:     Type type_;  // 型 (Type_INT か Type_CHAR)
 5:     int value_;  // 値
 6:  public:
 7:     Exp_constant(Type, t, int v) : type_(t), value_(v) {}  // コンストラクタ
 8:     ~Exp_constant() {}  // デストラクタ
 9:     Type type() const {return type_;}    // 型を読み出すメソッド
10:     int value() const {return value_;}   // 値を読み出すメソッド
11:     void print(std::ostream& os) const;  // 表示メソッド
12:  };
```

さて, このようにクラスを定義すると, 「定数」の抽象構文木は, 次のようなプログラムにより構築できる.

```
1:  Expression* c1 = new Exp_constant(Type_INT, 7);     // 7 の節点を作る
2:  Expression* c2 = new Exp_constant(Type_CHAR, 'x');  // 'x' の節点を作る
3:  c1->print(std::cout); std::cout<<std::endl;  // 7 の節点の表示
4:  c2->print(std::cout); std::cout<<std::endl;  // 'x' の節点の表示
```

これを実行すると,

```
7
'x'
```

という表示が得られる.

この表示を行う print の実装は次のようになる.

```
1:  void Exp_constant::print(std::ostream& os) const
2:  {
3:     switch(type()) {  // 型によって場合分け
4:        case Type_INT: os << value(); break;  // 整数型ならそのまま出力
5:        case Type_CHAR: os << '\'' << (char) value() << '\''; break;
6:           // 文字型なら変換 (cast) して出力.  single quote を前後につける
7:        default:  assert(0);  // 万一それ以外の型ならエラー
8:     }
9:  }
```

以上のプログラムは, ast.h と ast.cpp の雛型にサンプルとして書いてあるので, 確認せよ.

課題 5.1 定数 7, 'x' に対する抽象構文木を構築し, 表示するプログラム 5_01.cpp を作成し, 動作を確認せよ.

1. プログラムは次のように書けばよい.

   [5_01.cpp]

```
#include <iostream>
#include "ast.h"

int main(void)
{
  Expression* c1 = new Exp_constant(Type_INT, 7);
  Expression* c2 = new Exp_constant(Type_CHAR, 'x');
  c1->print(std::cout); std::cout<<std::endl;
  c2->print(std::cout); std::cout<<std::endl;
  return 0;
}
```

2. ast.cpp とともにコンパイル&リンクせよ.

```
g++ -g 5_01.cpp ast.cpp
```

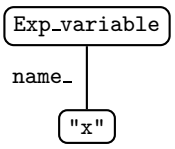
3. 実行し, 次の表示が得られることを確認せよ.

```
7
'x'
```

☆ レポートにプログラムは添付しなくてよい.

5.3.4 変数式 (Exp_variable)

式中に現れる x や abc 等の変数を表す. 下図は変数 x に対する抽象構文木である.



Exp_variable のクラスの仕様は次の通りである.

- 継承関係

  Expression の派生クラスである.

- メンバー

  1. std::string name_ … 変数名

- メソッド

  1. Exp_variable(const std::string& n)

     変数名を表す文字列 n を受け取り, name_ が n であるようなインスタンスを作るコンストラクタ.

  2. ~Exp_variable() … デストラクタ. 特に何もしない.
  3. const std::string& name() const … メンバー name_ の値を読み出す.
  4. void print(std::ostream& os) const … 表示メソッド. 変数名を表示する.

課題 5.2 Exp_variable を実装せよ (Exp_constant を参考にせよ.)

1. クラスの宣言を ast.h に追加せよ.
2. print の実装を ast.cpp に追加せよ.
3. 次のようなコード系列に対し, 実行を行え. 前課題の 5_01.cpp の main の中身だけ書き換えて, プログラム全体を作成し, これを 5_02.cpp とせよ.

```
Expression* v = new Exp_variable("n");
v->print(std::cout); std::cout<<std::endl;
```

   変数 n に対する抽象構文木が生成でき,

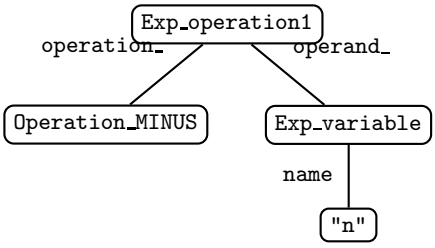
```
n
```

   のように, 変数名が表示されるようになればよい.

☆ レポートにはプログラム 5_02.cpp を添付せよ. ast.h と ast.cpp は, 一番最後に完成したものを添付する.

5.3.5 単項演算式 (Exp_operation1)

式中に現れる -n や +10 等, オペランドを 1 つだけ取る演算を表現する. 下図は変数 -n に対する抽象構文木である.



クラス Exp_operation1 の仕様は次の通りである.

- 継承関係

  Expression の派生クラスである.

- メンバー

1. Operator operation_

   演算. Operator は, ast.h 中に enum 型として次のように定義してあるので, 確認せよ.

```
enum Operator {
  Operator_PLUS,  // +
  Operator_MINUS, // -
  Operator_MUL,   // *
  Operator_DIV,   // /
  Operator_MOD,   // %
  Operator_LT,    // <
  Operator_GT,    // >
  Operator_LE,    // <=
  Operator_GE,    // =>
  Operator_NE,    // !=
  Operator_EQ,    // ==
};
```

2. Expression* operand_ … オペランドとなる式 (へのポインタ).

- メソッド

1. Exp_operation1(Operator op, Expression* ex)

   演算子が op, オペランドが ex のインスタンスを作るコンストラクタ.

2. ~Exp_operation1()

   デストラクタ. 下位の抽象構文木である operand_ を delete する. 次のようになる.

```
~Exp_operation1() {delete operand_;}
```

3. Operator operation() const … operation_ を読み出す.
4. const Expression* operand() const … operand_ を読み出す.
5. void print(std::ostream& os) const

   表示メソッド. 演算の優先順位にかかわらず括弧を表示するようにせよ. 例えば, 次のようになる. 3 行目の Operator_string は, Operator を表示用の文字列に変換する関数である. (例えば, Operator_MINUS を与えると "-" を返す. 実装は ast.cpp に書かれている.)

```
 1:  void Exp_operation1::print(std::ostream& os) const
 2:  {
 3:      os << "(" << Operator_string(operation());
 4:      if (operand()) {
 5:          operand()->print(os);
 6:      }
 7:      else {
 8:          os << "UNDEF";
 9:      }
10:      os << ")";
11:  }
```

課題 5.3 Exp_operation1 を実装せよ.

1. クラスの宣言を ast.h に追加せよ.
2. print の実装を ast.cpp に追加せよ.

3. 次のようなコード系列に対し, 実行を行え (作成したプログラムを 5_03.cpp とし, レポートに添付せよ).

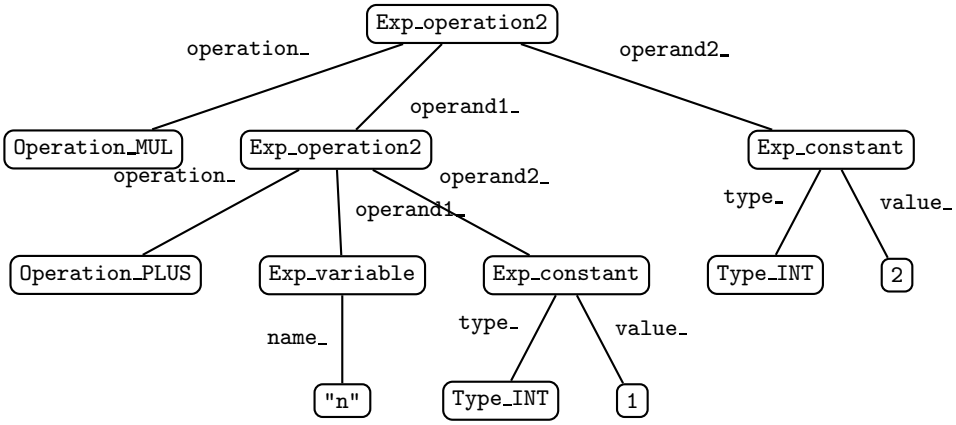
```
Expression* v = new Exp_variable("n");
Expression* o = new Exp_operation1(Operator_MINUS, v);
o->print(std::cout); std::cout<<std::endl;
```

4. 結果, 次のような表示が得られることを確認せよ.

```
(-n)
```

5.3.6 二項演算式 (Exp_operation2)

式中に現れる n+1 等, オペランドを 2 つ取る演算を表現する. 下図は変数 (n+1)*2 に対する抽象構文木である.



クラス Exp_operation2 の仕様は次の通りである.

- 継承関係

  Expression の派生クラスである.

- メンバー

  1. Operator operation_ … 演算.
  2. Expression* operand1_ … 第 1 オペランドとなる式 (へのポインタ).
  3. Expression* operand2_ … 第 2 オペランドとなる式 (へのポインタ).

- メソッド

  1. Exp_operation2(Operator op, Expression* ex1, Expression* ex2)

     演算子が op, オペランドが ex1, ex2 のインスタンスを作るコンストラクタ.

  2. ~Exp_operation2()

     デストラクタ. 下位の抽象構文木である operand1_, operand2_ を delete する.

  3. Operator operation() const … operation_ を読み出す.
  4. const Expression* operand1() const … operand1_ を読み出す.
  5. const Expression* operand2() const … operand2_ を読み出す.
  6. void print(std::ostream& os) const … 表示メソッド.

課題 5.4 Exp_operation2 を実装せよ.

1. クラスの宣言を ast.h に追加せよ.
2. print の実装を ast.cpp に追加せよ.
3. 次のようなコード系列に対し, 実行を行え (作成したプログラムを 5_04.cpp とし, レポートに添付せよ).

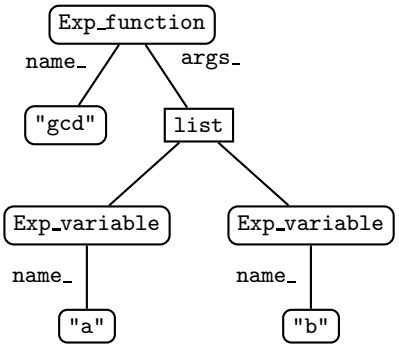
```
Expression* v = new Exp_variable("n");
Expression* c1 = new Exp_constant(Type_INT, 1);
Expression* o1 = new Exp_operation2(Operator_PLUS, v, c1);
Expression* c2 = new Exp_constant(Type_INT, 2);
Expression* o2 = new Exp_operation2(Operator_MUL, o1, c2);
o1->print(std::cout); std::cout<<std::endl;
o2->print(std::cout); std::cout<<std::endl;
```

4. 結果, 次のような表示が得られることを確認せよ. 演算の優先順位にかかわらず括弧を出力するようにせよ.

```
(n + 1)
((n + 1) * 2)
```

5.3.7 関数式 (Exp_function)

式中の gcd(a,b) 等の関数呼び出しを表現する. 下図は関数呼び出し gcd(a,b) に対する抽象構文木である.



クラス Exp_function の仕様は次の通りである.

- 継承関係

  Expression の派生クラスである.

- メンバー

  1. std::string name_ … 関数の名前.
  2. std::list<Expression*> args_ … 引数となる式 (へのポインタ) のリスト.

- メソッド

  1. Exp_function(const std::string& nm, const std::list<Expression*>& args)

     関数名が nm, 引数リストが args であるようなインスタンスを作るコンストラクタ.

  2. ~Exp_function()

     デストラクタ. args_ のリスト中の式の構文木を全て消去する. ast.h のクラスの宣言中では

     ```
     ~Exp_function();
     ```

     とだけ書き, 実装は ast.cpp 中に書く. 例えば, 次のようになる.

```
Exp_function::~Exp_function()
{
  for (std::list<Expression*>::const_iterator it = args_.begin();
      it != args_.end(); it++) {
    delete *it;
  }
}
```

3. `const std::string& name() const` … `name_` を読み出す.
4. `const std::list<Expression*>& args() const` … `args_` を読み出す.
5. `void print(std::ostream& os) const` … 表示メソッド.

課題 5.5 `Exp_function` を実装せよ.

1. クラスの宣言を `ast.h` に追加せよ.

   `std::list` を用いるので, `<list>` を include せよ.

2. デストラクタと `print` の実装を `ast.cpp` に追加せよ.

   引数リストの走査には, `list<Expression*>::iterator` ではなく, `list<Expression*>::const_iterator` を用いよ.

3. 次のようなコード系列に対し, 実行を行え (作成したプログラムを `5_05.cpp` とし, レポートに添付せよ).

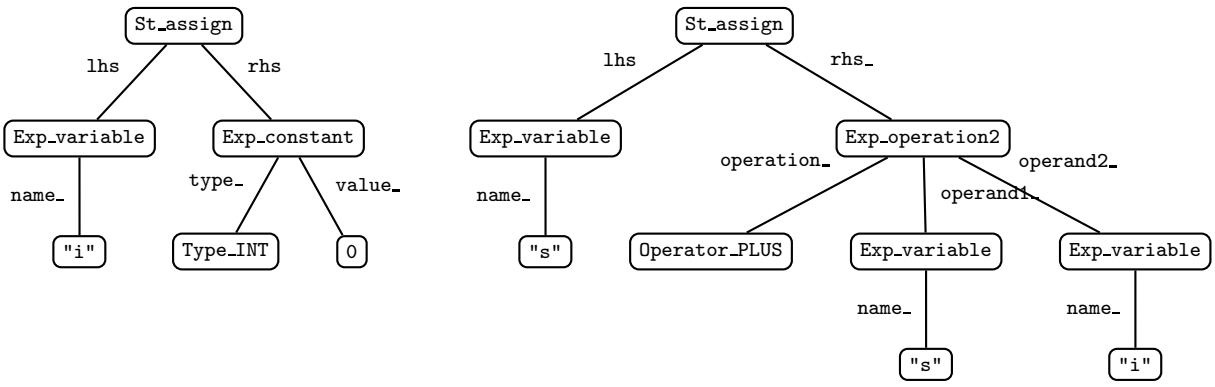
```
Expression* v1 = new Exp_variable("a");
Expression* v2 = new Exp_variable("b");
std::list<Expression*> arglist;
arglist.push_back(v1);
arglist.push_back(v2);
Expression* f = new Exp_function("gcd", arglist);
f->print(std::cout); std::cout<<std::endl;
```

4. 結果, 次のような表示が得られることを確認せよ.

```
gcd(a,b)
```

5.3.8 代入文 (St_assign)

下図はそれぞれ `i = 0;` と `s = s + i;` に対する抽象構文木である.



`St_assign` を定義する前に, まず「文」を表す抽象基底 `Statement` を次のように定義する.

```
 1:  class Statement
 2:  {
 3:  public:
 4:     Statement() {}  // コンストラクタ
 5:     virtual ~Statement() {}   // デストラクタ
 6:     virtual void print(std::ostream& os, int indent=0) const = 0;  //表示
 7:  private:
 8:     Statement(const Statement&);  // コピーコンスタラクタは禁止
 9:     Statement& operator=(const Statement&);  // 代入演算は禁止
10:  };
```

– 表示メソッド print のデフォルト引数 indent は, 表示の際に何段インデントするかを指定するものである. デフォルトの indent=0 はインデントせず, 例えば indent=2 では 2 段 (現在の実装ではスペース 4 個) のインデントを文の前に出力する.

St_assign は Statement の派生クラスとして, 次のように定義できる.

```
 1:  class St_assign : public Statement
 2:  {
 3:  private:
 4:     Exp_variable* lhs_;  // 左辺 (変数)
 5:     Expression* rhs_;    // 右辺 (式)
 6:  public:
 7:     St_assign(Exp_variable* lexp, Expression* rexp)  // コンストラクタ
 8:        :  lhs_(lexp), rhs_(rexp) {}
 9:     ~St_assign()  // デストラクタ
10:     {
11:        delete lhs_;   // 子の抽象構文木は delete する
12:        delete rhs_;   // 子の抽象構文木は delete する
13:     }
14:     const Exp_variable* lhs() const { return lhs_; }   // 左辺の読み出し
15:     const Expression* rhs() const { return rhs_; }     // 右辺の読み出し
16:     void print(std::ostream& os, int indent=0) const;  // 表示
17:  };
```

表示メソッド print の実装は例えば, 次のようになる.

```
 1:  void St_assign::print(std::ostream& os, int indent) const
 2:  {
 3:     os << tab(indent);    // インデント (1 段につきスペース 2 個) をつける
 4:     if (lhs()) {          // 左辺が NULL でなければ
 5:        lhs()->print(os);  // 左辺を表示
 6:     }
 7:     else {
 8:        os << "UNDEF";     // 左辺が NULL なら取り敢えず UNDEF と表示
 9:     }
10:     os << " = ";
```

```
11:     if (rhs()) {          // 右辺が NULL でなければ
12:        rhs()->print(os);  // 右辺を表示
13:     }
14:     else {
15:        os << "UNDEF";
16:     }
17:     os << ";" << std::endl;
18:  }
```

- 3 行目: tab(int indent) は, 2×indent 個の空白 (std::string) を返す関数であり, ast.cpp 中に定義されている.
- 5 行目: 左辺の表示は lhs() の print メソッドを呼ぶだけだが, その前に lhs() が NULL でないことを確認しておく必要がある.
- 11 行目も同様.

課題 5.6 St_assign を実装せよ.

1. クラスの宣言を ast.h に追加せよ.
2. print の実装を ast.cpp に追加せよ.
3. 次のようなコード系列に対し, 実行を行え. (作成したプログラムを 5_06.cpp とし, レポートに添付せよ).

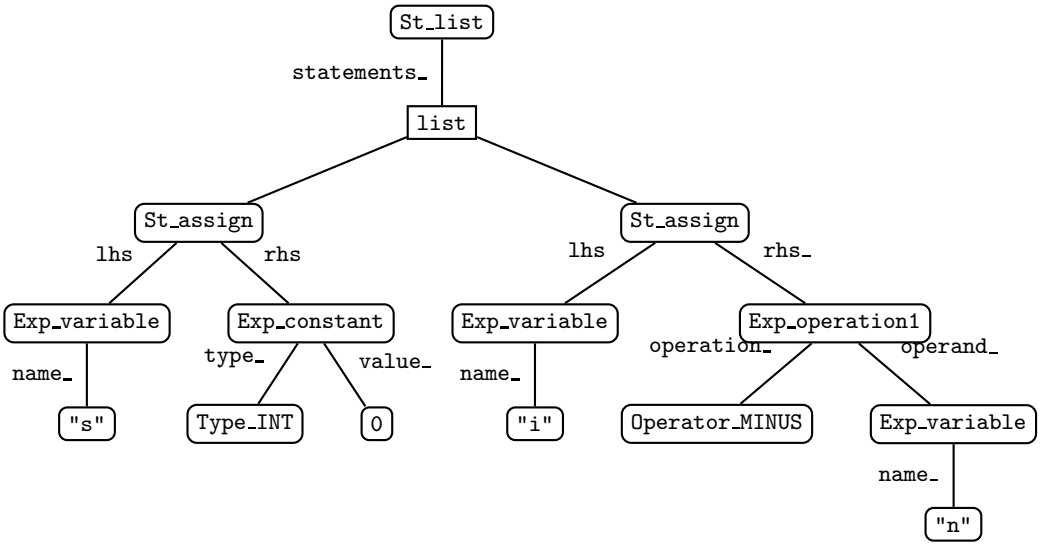
```
Exp_variable* v1 = new Exp_variable("i");
Expression* c1 = new Exp_constant(Type_INT, 0);
Statement* s1 = new St_assign(v1,c1);
s1->print(std::cout);
s1->print(std::cout, 1);
s1->print(std::cout, 2);
```

4. 結果, 次のような表示が得られることを確認せよ. print の第 2 引数に指定されたインデントの段数に応じたインデントが行われる.

```
i = 0;
  i = 0;
    i = 0;
```

5.3.9 文の並び (St_list)

連続する複数の文をまとめて表現する抽象構文木である. その名の通り文のリストであり, 例えば {s=0; i=-n;} という連続する 2 つの文は, 次のように表現される.



クラス St_list の仕様は次の通りである.

- 継承関係

  Statement の派生クラスである.

- メンバー

  1. std::list<Statement*> statements_ … 文のリスト.

- メソッド

  1. St_list(const std::list<Statement*>& li)

     文のリストが li であるようなインスタンスを作るコンストラクタ.

  2. ~St_list() … デストラクタ. リスト中の式の構文木を全て消去する.
  3. const std::list<Statement*>& statements() const … statements_ を読み出す.
  4. void print(std::ostream& os, int indent=0) const … 表示メソッド.

課題 5.7 St_list を実装せよ.

1. クラスの宣言を ast.h に追加せよ.
2. デストラクタと print の実装を ast.cpp に追加せよ.
3. 次のようなコード系列に対し, 実行を行え (作成したプログラムを 5_07.cpp とし, レポートに添付せよ).

[List 5.1]

```
// 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);
```

```
// St_list を構成
std::list<Statement*> slist;
slist.push_back(s1);
slist.push_back(s2);
Statement* s3 = new St_list(slist);
// 表示
s3->print(std::cout,1);
```

4. 次のような表示が得られることを確認せよ. 特に, リスト中の全ての文に対してインデントが付けられるようにせよ.

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

☆☆☆ 課題の前半はここまで ☆☆☆

5.3.10 if 文 (St_if)

cond_, then_, else_ という 3 つのメンバーを持ち, それぞれ, 条件を表す式, 条件が成立するときに実行する文, 条件が成立しないときに実行する文を表す. 例えば

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

という if 文があるとき

cond_ は i<0

then_ は s = s - i;

else_ は s = s + i;

を指す.
クラス St_if の仕様は次の通りである.

- 継承関係

  Statement の派生クラスである.

- メンバー

  1. Expression* cond_ … 条件式
  2. Statement* then_ … then 部の文
  3. Statement* else_ … else 部の文

- メソッド

  1. St_if(Expression* cond, Statement* then, Statement* els)

条件式が cond, then 部の文が then, else 部の文が els [6] であるようなインスタンスを作るコンストラクタ.

2. ~St_if()

   デストラクタ. cond_, then_, else_ を delete する.

3. const Expression* condition() const … cond_ を読み出す.
4. const Statement* then_part() const … then_ を読み出す.
5. const Statement* else_part() const … else_ を読み出す.
6. void print(std::ostream& os, int indent=0) const … 表示.

課題 5.8 St_if を実装せよ.

1. クラスの宣言を ast.h に追加せよ.
2. print の実装を ast.cpp に追加せよ.
3. 次の if 文を表す抽象構文木を構築して表示するプログラム 5_08.cpp を作成せよ.

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

この if 文の抽象構文木は以降の演習で再利用するので, この抽象構文木を構築する部分を関数 Statement* make_if() として独立させよ. つまり, main() の中に構築用の文を直接書くのではなく, 次のように main() から make_if() を呼び出す形にせよ.

```
Statement* make_if() {
  // cond 部の木の生成
  …
  Expression* cond = …;

  // then 部の木の生成
  …
  Statement* then = …;

  // else 部の木の生成
  …
  Statement* els = …;  // else は予約語なので変数名には使えない

  return new St_if(cond, then, els);
}
```

[6] else は C/C++ の予約語なので変数名として使えない.

```
int main()
{
  Statement *s = make_if();
  s->print(std::cout);
  return 0;
}
```

4. 正しく表示が行われることを確認せよ. インデントや { } の改行位置は, 各自の趣味で設定してよい.

5.3.11 while 文 (St_while)

while ループを表現する抽象構文木である. cond_, body_ という 2 つのメンバーを持ち, それぞれ, 条件を表す式, ループの本体の文を表す. 例えば

```
while (i<n) {
  s=s+i;
  i=i+1;
}
```

という while 文があるとき

cond_ は i<n

body_ は {s=s+i; i=i+1;}

である.

クラス St_while の仕様は次の通りである.

- 継承関係

  Statement の派生クラスである.

- メンバー

  1. Expression* cond_ … 条件式
  2. Statement* body_ … ループ本体の文

- メソッド

  1. St_while(Expression* cond, Statement* body)

     条件式が cond, 本体の文が body であるようなインスタンスを作るコンストラクタ.

  2. ~St_while() … デストラクタ. cond_, body_ を delete する.
  3. const Expression* condition() const … cond_ を読み出す.
  4. const Statement* body() const … body_ を読み出す.
  5. void print(std::ostream& os, int indent=0) const … 表示メソッド.

課題 5.9 St_while を実装せよ.

1. クラスの宣言を ast.h に追加せよ.
2. print の実装を ast.cpp に追加せよ.
3. 次の while 文を表す抽象構文木を構築して表示するプログラム 5_09.cpp を作成せよ.

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

while 文の中の if 文は前課題と同じものなので, 作成した `make_if()` を再利用せよ. `body_` は, この `if` 文と `i=i+1;` という 2 つの文から成る「文の並び」(`St_list`) であることに注意. この while 文の抽象構文木自身, 以降の演習で再利用するので, この抽象構文木を構築する部分を関数 `Statement* make_while()` として独立させよ.

4. 正しく表示が行われることを確認せよ.

5.3.12 return 文 (St_return)

return 文を表現する抽象構文木で, 返り値を表す式 (へのポインタ) を格納する `value_` をメンバーに持つ. 例えば

```
return s;
```

において, 式 `s` が返り値である.
クラス `St_return` の仕様は次の通りである.

- 継承関係

  `Statement` の派生クラスである.

- メンバー

  1. `Expression* value_` … 返り値の式

- メソッド

  1. `St_return(Expression* value)`

     返り値を表す式が `value` であるようなインスタンスを作るコンストラクタ.

  2. `~St_return()` … デストラクタ. `value_` を delete する.
  3. `const Expression* value() const` … `value_` を読み出す.
  4. `void print(std::ostream& os, int indent=0) const` … 表示メソッド.

課題 5.10 `St_return` を実装せよ.

1. クラスの宣言を `ast.h` に追加せよ.
2. `print` の実装を `ast.cpp` に追加せよ.
3. 次の return 文を表す抽象構文木を構築して表示するプログラム `5_10.cpp` を作成せよ.

   ```
   return s;
   ```

4. 正しく表示が行われることを確認せよ.

5.3.13 関数呼び出し文 (St_function)

putint(a); 等の関数を呼び出しを表現する．先に定義したクラス Exp_function とほぼ同じである．違いは Exp_function が「式」だったのに対し, St_function は「文」であるという点だけである．
クラス St_function の仕様は次の通りである．

- 継承関係

  Statement の派生クラスである．

- メンバー

  1. std::string name_ … 関数の名前．
  2. std::list<Expression*> args_ … 引数となる式 (へのポインタ) のリスト．

- メソッド

  1. St_function(const std::string& nm, const std::list<Expression*>& args)

     関数名が nm, 引数リストが args であるようなインスタンスを作るコンストラクタ．

  2. ~St_function()

     デストラクタ．args_ のリスト中の式の構文木を全て消去する．

  3. const std::string& name() const … name_ を読み出す．
  4. std::list<Expression*>& args() const … args_ を読み出す．
  5. void print(std::ostream& os, int indent=0) const

     表示メソッド．Exp_function の print に対し, 先頭のインデントと末尾のセミコロン (;) が付加される点が異なる．

このように, St_function は Exp_function をわずかだけ拡張して型を変えたものだが, このようなクラスを効率よく実装する手法には 2 通りがある．一つは継承である．しかし, St_function を Exp_function の派生クラスとすると, 意味の異なる多重の継承関係が発生してしまい, あまり好ましい設計とは言えない．もう一つは, 「包含」(containment) あるいは「合成」(composition) と呼ばれる方法である．これは, 元になるクラス (ここでは Exp_function) を拡張された新しいクラス (ここでは St_function) のメンバーにすることにより定義する方法である [7]．ここではこの方法を用いる．

具体的には, 次のようにクラスを定義すればよい．

```
 1:  class St_function : public Statement
 2:  {
 3:  private:
 4:     Exp_function function_;  // Exp_function を包含
 5:  public:
 6:     // 実装は function_ のメソッドを用いて定義
 7:     St_function(const std::string& name,
 8:           const std::list<Expression*>& args)
 9:        :  function_(name, args) {}
10:     ~St_function() {}  // デストラクタは連鎖するので特に何も書かない
11:     const std::string& name() const{return function_.name();}
12:     const std::list<Expression*>& args()
13:        const { return ???;}  // ここは自分で考えよ
14:     void print(std::ostream& os, int indent = 0) const;
15:  };
```

[7] 継承がサポートされない言語で継承を疑似的に実装する手法としても用いられている．ポリモーフィズムを利用する以外の継承は避け, クラスの拡張などは包含により実装すべしとの立場がある．

St_function::print の実装も, Exp_function::print のメソッドを利用して行う. 具体的には,

```
インデントを出力;
function_ の print メソッドを呼び出す;
セミコロンと改行を出力;
```

とすればよい.

課題 5.11 St_function を実装せよ.

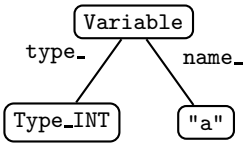
1. クラスの宣言を ast.h に追加せよ.
2. print の実装を ast.cpp に追加せよ.
3. 次の関数呼び出し文の抽象構文木を構築して表示するプログラム 5_11.cpp を作成せよ.

```
putint(a);
```

4. 正しく表示が行われることを確認せよ.

5.3.14 変数宣言 (Variable)

int a や char ch 等の変数宣言を表す抽象構文木のクラスである. 例えば, int a に対する抽象構文木は次のようになる.



クラス Variable の仕様は次の通りである.

- 継承関係 … なし[8]
- メンバー
  1. Type type_ … 変数の型 (Type_INT か Type_CHAR)
  2. std::string name_ … 変数名
- メソッド
  1. Variable(Type type, const std::string& name)

     型が type, 名前が name であるようなインスタンスを作るコンストラクタ.
  2. ~Variable() … デストラクタ. 特に何もしない.
  3. Type type() const … type_ を読み出す.
  4. const std::string& name() const … name_ を読み出す.
  5. void print(std::ostream& os) const … 表示. 単に型と変数名を表示するだけ. (インデントもしないし, セミコロンも付加しない.)

課題 5.12 Variable を実装せよ.

1. クラスの宣言を ast.h に追加せよ.
2. print の実装を ast.cpp に追加せよ.

   ☆ print 内部で改行を行わないこと. でないと, (次項の) 関数の引数の宣言

   ```
   int gcd(int a, int b)
   ```

   の部分の抽象構文木を表示すると

[8] Statement の派生クラスではないことに注意

```
int gcd(int a
, int b
)
```

などということになってしまう.

3. 次の変数宣言を表す抽象構文木を構築して表示するプログラム 5_12.cpp を作成せよ.

```
int a
```

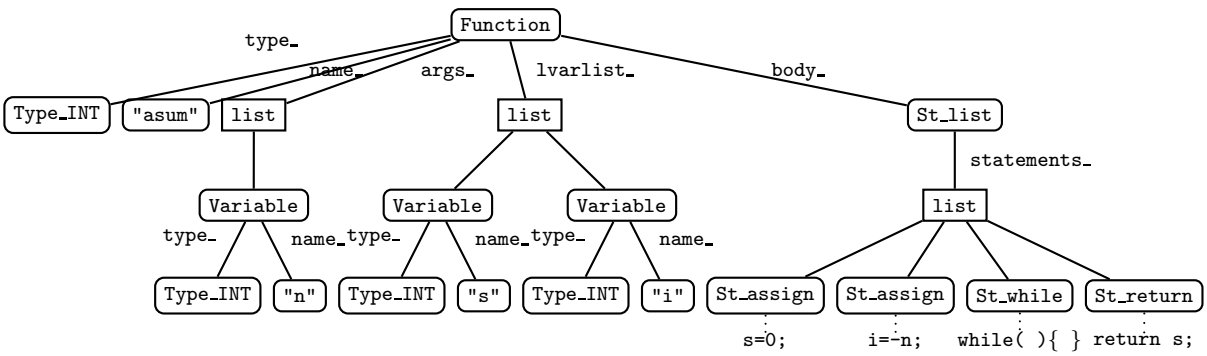
4. 正しく表示が行われることを確認せよ.

5.3.15 関数宣言 (Function)

関数の宣言全体を表す. 例えば, 次のプログラムに対する抽象構文木は下のようになる.

[List 5.2]

```
 1:  int asum(int n)
 2:  {
 3:     int s;
 4:     int i;
 5:
 6:     s = 0;
 7:     i = -n;
 8:     while (i<=n) {
 9:        if (i<0) {
10:           s = s - i;
11:        }
12:        else {
13:           s = s + i;
14:        }
15:        i = i + 1;
16:     }
17:     return s;
18:  }
```



クラス Function の仕様は次の通りである.

- 継承関係 … なし
- メンバー

1. Type type_ … 関数の型.
2. std::string name_ … 関数の名前.
3. std::list<Variable*> args_ … 引数の宣言のリスト.
4. std::list<Variable*> lvarlist_ … ローカル変数の宣言のリスト.
5. Statement* body_ … 関数の本体

- メソッド

1. Function(Type type, const std::string& name,
   const std::list<Variable*>& args,
   const std::list<Variable*>& lvarlist,
   Statement* body)

   型が type, 関数名が name, 引数宣言のリストが args, ローカル変数の宣言のリストが lvarlist, 本体が body であるようなインスタンスを作るコンストラクタ.
2. ~Function()

   デストラクタ. args_, lvarlist_, body_ の構文木を全て消去する.
3. Type type() const … type_ を読み出す.
4. const std::string& name() const … name_ を読み出す.
5. const std::list<Variable*>& args() const … args_ を読み出す.
6. const std::list<Variable*>& lvarlist() const … lvarlist_ を読み出す.
7. const Statement* body() const … body_ を読み出す.
8. void print(std::ostream& os) const … 表示メソッド.

課題 5.13 Function を実装せよ.

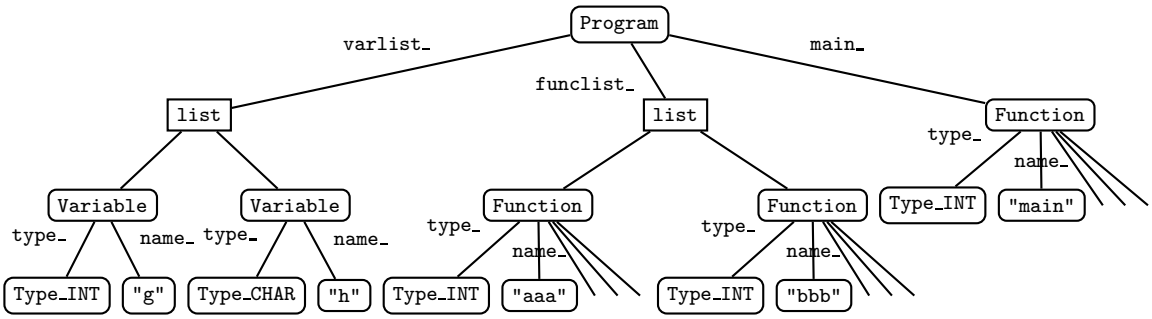
1. クラスの宣言を ast.h に追加せよ.
2. デストラクタと print の実装を ast.cpp に追加せよ.
3. [List 5.2] の関数の構文木を構築するプログラム 5_13.cpp を作成せよ. (以前の課題で作成した make_while を再利用せよ.) この関数宣言の抽象構文木を以降の演習で再利用するので, この抽象構文木を構築する部分を関数 Function* make_function_asum() として独立させよ.
4. 実行して正しい表示が得られることを確認せよ.

5.3.16 プログラム全体 (Program)

プログラム全体を表す. 例えば, 次のプログラムに対する抽象構文木は下のようになる.

```
 1:  int g;
 2:  char h;
 3:
 4:  int aaa(int a)
 5:  {
 6:  …
 7:  }
 8:
 9:  int bbb()
10:  {
11:  …
12:  }
```

```
13:
14:  int main()
15:  {
16:  …
17:  }
```



クラス Program の仕様は次の通りである.

- 継承関係 … なし
- メンバー
  1. `std::list<Variable*> varlist_` … グローバル変数の宣言のリスト.
  2. `std::list<Function*> funclist_` … 関数の宣言のリスト.
  3. `Function* main_` … main 関数.

     ☆ 構文上は main 関数も他の関数と変わらないので, `funclist_` の要素にしてしまうというのも一つの方法だが, 実質的に main 関数は特別な意味を持つので, ここでは分離してデータを保持することにする.
- メソッド
  1. `Program( const std::list<Variable*>& varlist,`
     `const std::list<Function*>& funclist, Function* main)`

     変数の宣言のリストが varlist, 関数の宣言のリストが funclist, main 関数が main であるようなインスタンスを作るコンストラクタ.
  2. `~Program()`

     デストラクタ. 各メンバーの抽象構文木を全て消去する.
  3. `const std::list<Variable*>& varlist() const` … `varlist_` を読み出す.
  4. `const std::list<Function*>& funclist() const` … `funclist_` を読み出す.
  5. `const Function* main() const` … `main_` を読み出す.
  6. `void print(std::ostream& os) const` … 表示メソッド.

課題 5.14 Program を実装せよ.

1. クラスの宣言を ast.h に追加せよ.
2. デストラクタと print の実装を ast.cpp に追加せよ.
3. 次の [List 5.3] の関数の構文木を構築するプログラム 5_14.cpp を作成せよ.
   (前の課題で作成した make_function_asum を再利用せよ.)
4. 実行して正しい表示が得られることを確認せよ.

[List 5.3]

```
 1:  int g;
 2:
 3:  int asum(int n)
 4:  {
 5:     int s;
 6:     int i;
 7:
 8:     s = 0;
 9:     i = -n;
10:     while(i<=n) {
11:        if (i<0) {
12:           s = s - i;
13:        }
14:        else {
15:           s = s + i;
16:        }
17:        i = i + 1;
18:     }
19:     return s;
20:  }
21:
22:  int main()
23:  {
24:     int a;
25:     g = 3;
26:     a = asum(g);
27:     putint(a);
28:  }
```

課題 5.15 講義ホームページより factor.cpp をダウンロードし, ast.cpp とリンクして実行し, 下のような結果が得られることを確認せよ.

- インデントがきれいにつかなくても OK とする.
- この課題に関するプログラムはレポートに添付しなくてよいが, 最後に完成した ast.h と ast.cpp をレポートに添付せよ.

[List 5.4]

```
 1:  char separator;
 2:
 3:
 4:  int factor(int n)
 5:  {
 6:     int d;
 7:     d = 2;
 8:     while (((d * d)  <= n)) {
 9:        if (((n % d) == 0)) {
10:           putint(d);
11:           putchar(separator);
12:           n = (n / d);
13:        }
14:        else {
15:           d = (d + 1);
16:        }
17:     }
18:     putint(n);
19:     putchar('\n');
20:  }
21:
22:
23:  int main()
24:  {
25:     int n;
26:     putchar('n');
27:     putchar('=');
28:     n = getint();
29:     separator = '*';
30:     factor(n);
31:  }
```

Nagisa ISHIURA