システムソフトウェア特論'19 # 13 オブジェクト指向の実装

久野 靖 (電気通信大学)

2019.1.9

1　オブジェクト指向言語の実装の留意点

前回までで一般的な手続き型言語の実装については一通り扱い終わりましたが、今日では手続き型言語の中でもオブジェクト指向言語が広く使われるようになっています。そして、オブジェクト指向言語の実装にはこれまでに取り上げて来た手続き型言語に無い次のような留意点があります。

(a) オブジェクト (インスタンス) の生成 — 複数の変数 (インスタンス変数) 群と手続き群 (メソッド) が組になったインスタンスが作り出せる。

(b) メソッド呼び出し — インスタンスを指定してメソッドを呼び出すと、そのインスタンスに付随するメソッドが呼び出される (動的分配)。

(c) インスタンス変数アクセス — インスタンスメソッド内から変数にアクセスすると、そのインスタンスのインスタンス変数がアクセスできる。

これらのうち (a) は抽象データ型 (内部で複雑な構造を持つような値を複数作り出す機能) を実現できるという意味を持ちます。(b) は様々なオブジェクトに対し同一のコード (呼び出し) が適用でき、オブジェクトの種類が多くても記述の複雑さを抑制できるという意味を持ちます。(c) は情報隠蔽ないしカプセル化 (オブジェクト内部のデータを外部から見られたり変更されたりすることを防ぐ) が可能になるという意味を持ちます。

そして、ここから先は言語のデザインによって有無が違って来ますが、次のような機能を持つ言語も複数見られます。

(d) クラス (オブジェクトの形を定義する「型枠」のような構文単位) が定義でき、クラスに基づいて複数のオブジェクト (インスタンス) を生成する。

(e) クラス間またはオブジェクト間に継承関係が定義でき、それに基づいて「あるオブジェクトに類似した (しかし同一ではなく違いもある) オブジェクト」を定義できる。

(f) クラスやメソッドについて静的な型が定まっていて、それに基づき型検査がおこなえる。

(g) コンテナ型 (配列などのように他のオブジェクトを格納することを目的としたオブジェクトの型) など、さまざまな種別のオブジェクトを扱うようなオブジェクトに対して、扱うオブジェクトの種類 (型) をパラメタとして与えて全体の型を定める型パラメタ機能を持つ。

これらは後の方にいくほど機能的にも理論的にも複雑になります。ここでは具体的な実装例を見るため、まず (a)〜(d) について取り上げ、そのあと (e)〜(f) について簡単に説明します。(g) の型パラメタについては内容が多くなりすぎるため、ここでは扱いません。

2　インスタンス変数と動的分配の実装

2.1　C 言語上での実装記述

ここまででの言語処理系の実装例では翻訳系の目的コードとして機械語 (アセンブリ言語) を採用して来ましたが、ここから先はコードが複雑になるので C 言語を目的コードとして使用します。

C 言語を目的コードとして使用することは、C コンパイラによる翻訳が必要になるという手間や、C 言語の型検査をパスするようにコードを生成する必要があるという手間が掛かりますが、一方で次のような利点もあります。

- C コンパイラは多くのシステムで利用可能なので、生成コードを多くのシステムで動かせる。
- 実用レベルの C コンパイラは最適化に力を入れているため、自前で最適化を作成するより高品質な (性能のよい) コードが結果的に得られることが多い。
- C の方がアセンブリ言語より読みやすいため、生成コードの検討が容易である。
- 生成コードに間違いがあった時に、C コンパイラの型検査によって検出できることが多くある。

2.2 データ構造の設計

以下では、前節の (a)〜(c) を実現するようなデータ構造を検討し、続いてそれを C 言語のデータ構造として表現します。まず最初に一番複雑そうな (b) から検討しましょう。(b) の本質は、あるオブジェクトについて、そのオブジェクトが持つメソッドの名前を指定して呼び出す、というところにあります。これを実現するため、図 1 のような構造を使用します。これをメソッドを検索する表であることから「メソッドテーブル」と呼びます。

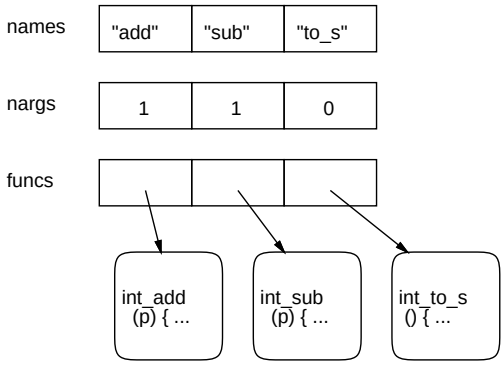

図 1: メソッドテーブルの実装例

使い方ですが、メソッド名を文字列として与えられたら、配列 `names` の上で探索して該当位置をみつけ、その対応位置にある配列 `funcs` の関数のコードを呼び出せばよいわけです。C 言語には「関数へのポインタ」を扱う機能があるため、これが素直に書けます。図 1 は整数オブジェクト用の表を想定しているので、各関数きは整数オブジェクト用ですが、実数オブジェクト用の表であれば実数オブジェクト用の関数へのポインタを入れておきます。

なお、配列 `nargs` はそれぞれのメソッドが (オブジェクト自身に加えて) いくつパラメタを受け取るかを表しています。整数オブジェクトを文字列に変換する `to_s` は「`i.to_s`」のようにパラメタを受け取りませんが、足し算は「`i.add(3)`」のようにパラメタが 1 つ必要です。

実際には、オブジェクトごとにそのオブジェクト用の表を使う必要があるので、オブジェクト 1 つごとにメソッドテーブルへのポインタを持つようにします。メソッドテーブルは 3 つの配列なので、それをレコードにすればポインタは 1 つでいいのですが、ここではあとで読むコードが簡単になるため、3 つのポインタでそれぞれの配列を指しています。

そして、整数オブジェクトであれば (`ty_int` という型名をつけていますが)、先頭にその種別 (整数は 1 としました) を表すタグ (番号)、その次にメソッドの個数 (検索時に上限が必要なので)、その後に 3 つの配列へのポインタがあり、そのさらに下にそのオブジェクトのインスタンス変数を必要なだけ取ります。整数オブジェクトであればその整数の値があれば十分なので `val` というフィールドを 1 つ持たせました。

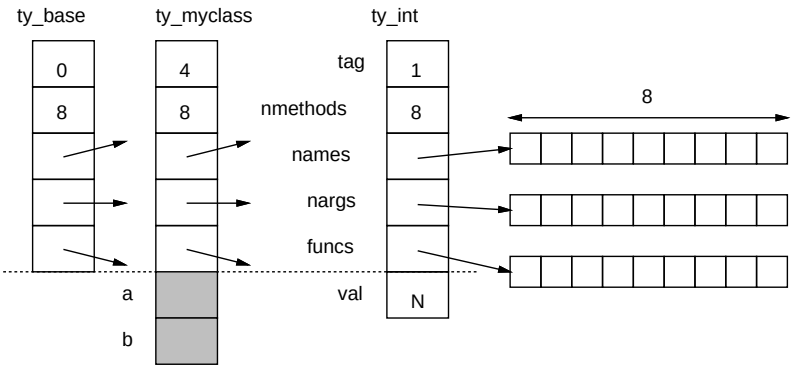

図 2: オブジェクトの構造

整数オブジェクトはすべてこの同じ形ですが、オブジェクトの種類が違えば形が違います (そして指すメソッドテーブルも違います)。たとえば myclass にインスタンス変数 a、b があれば、それはオブジェクトの末尾に置かれるのでその部分が整数と違います。インスタンス変数がまったく無ければ ty_base のような形になります。

ここで重要なのは、点線から上の部分はどのオブジェクトでも共通であり、同じコードで扱えるという点です。C 言語でいえば、ty_int へのポインタでも ty_myclass へのポインタでも ty_base へのポインタにキャストできますから、そうしてから扱うことでどの種類のオブジェクトでも同じに処理が書けます。

2.3 実行時ライブラリのソース

前節で紹介した方針によるオブジェクトとして nil クラス (値がないことを表す)、int クラス (整数)、str クラス (文字列) の 3 種類のクラスをまず手で作成します。後で作成する言語ではこれらは**組み込みクラス** (bulit-in class) として「既に存在する」状態で始まります。そのようなものが無いと言語として成り立たないことは明らかですから。

まず、ヘッダファィルを示します。これはこのライブラリを使う C 言語ソースで読み込むことを意図しています。まず 3 種類のクラスに対応するタグ値と、整数および実数オブジェクトを生成する関数を宣言します (nil は 1 つだけインスタンスを用意すればいいので生成は不要)。オブジェクト値は C 言語上では「任意のポインタ」を表す void*型として扱います。

```
// objlib.h -- header file for simple O-O runtime.
#define TAG_NIL 0
#define TAG_INT 1
#define TAG_STR 2
void *int_new(int v);
void *str_new(char *s);
```

続いて、3 種類のレコード型を定義します。構造は先に説明した通りです。nil 値はその 1 つだけのインスタンスをグローバル変数として用意し、そのアドレスを使うようになります。

```
typedef struct {
  int tag, nmethods; void* (**funcs)(); char* *names; int *nargs;
} ty_base;
extern ty_base nil_val;
typedef struct {
  int tag, nmethods; void* (**funcs)(); char* *names; int *nargs;
  int val;
```

```
} ty_int;
extern ty_int int_1_val, int_0_val;
typedef struct {
  int tag, nmethods; void* (**funcs)(); char* *names; int *nargs;
  char *str;
} ty_str;
```

最後に、メソッドを呼び出すための関数 send0、send1 と条件判断の際に呼び出す関数 tst の宣言があります。これらの内容は後で読みます。

```
void *send0(void *p, char *name);
void *send1(void *p, char *name, void *q);
int tst(void *p);
```

ではここから、ライブラリ本体のソースです。必要な include から始まります。

```
// objlib.c --- runtime for simple O-O language.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "objlib.h"
```

まず nil オブジェクトの実装です。メソッドとしては to_i、to_s、print、println があります。to_i は整数の 0、to_s は文字列の"nil"を返します。残りり 2 つは nil と出力するもので、改行の有無が違うだけです。

```
void *nil_to_i(void *p) {
  return &int_0_val;
}
void *nil_to_s(void *p) {
  return str_new("nil");
}
void *nil_print(void *p) {
  printf("nil"); return p;
}
void *nil_println(void *p) {
  printf("nil\n"); return p;
}
char *nil_names[] = {"to_i", "to_s", "print", "println"};
int nil_nargs[] = {0,0,0,0};
void* (*nil_funcs[])() = {nil_to_i, nil_to_s, nil_print, nil_println};

ty_base nil_val = {
  TAG_NIL, sizeof(nil_nargs)/sizeof(int), nil_funcs, nil_names, nil_nargs
};
```

そして末尾に names、nargs、funcs の配列を用意し、1 つだけある nil オブジェクトはタグ、メソッド数、3 つの配列へのポインタで初期化します。

次は順番を入れ換えて文字列を読みましょう (この方が少しシンプル)。まず文字列への変換 to_s は、すでに自身が文字列オブジェクトなので自分を返すだけです。to_i は atoi を呼んで文字列を整

数にし、整数オブジェクトを返します。print、println は値の文字列を打ち出します。どのメソッドも最初の引数として自オブジェクトへのポインタを受け取ります。add は 2 つの文字列の連結 add で、これはバッファに 2 つの文字列オブジェクトの文字列値をくっつけた文字列を構成した後、その文字列を内容とする新たなオブジェクト値を返します prompt は文字列をプロンプトとして出力し、入力した文字列を返します。

```
void *str_to_s(void *p) {
  return p;
}
void *str_to_i(void *p) {
  return int_new(atoi(((ty_str*)p)->str));
}
void *str_print(void *p) {
  printf("%s", ((ty_str*)p)->str); return p;
}
void *str_println(void *p) {
  printf("%s\n", ((ty_str*)p)->str); return p;
}
void *str_add(void *p, void *q) {
  char buf[100];
  sprintf(buf, "%s%s", ((ty_str*)p)->str, ((ty_str*)send0(q, "to_s"))->str);
  return str_new(buf);
}
void *str_prompt(void *p) {
  char buf[100];
  printf(((ty_str*)p)->str); fgets(buf, 100, stdin);
  buf[strlen(buf)-1] = '\0'; return str_new(buf);
}
char *str_names[] = {"to_i", "to_s", "print", "println", "add", "prompt"};
int str_nargs[] = {0,0,0,0,1,0};
void* (*str_funcs[])() = {str_to_i, str_to_s, str_print, str_println,
  str_add, str_prompt};
void *str_new(char *s) {
  ty_str *p = (ty_str*)malloc(sizeof(ty_str));
  p->tag = TAG_STR; p->nmethods = sizeof(str_nargs)/sizeof(int);
  p->funcs = str_funcs; p->names = str_names; p->nargs = str_nargs;
  p->str = (char*)malloc(strlen(s)+1); strcpy(p->str, s);
  return (void*)p;
}
```

3 つの配列 (メソッドテーブル) は先と同様。そして str_new ですが、まずレコード領域を malloc で割り当て、それぞれのフィールドを初期化していきます。文字列値については、値の文字列が入る領域を割り当て、そこに値をコピーします (元のポインタをそのまま使っていると値が変化してしまう恐れがあるので)。初期化が終わったらレコードのポインタを返します。

最後に整数です。to_i は自身をそのまま返し、to_s は sprintf でバッファに文字列を出力し、文字列オブジェクトにして返します。print、println は数値を出力します。その先ですが、+-*/%の演算はみなメソッド名と演算を除けば同じ形になるので、その同じ形をマクロで定義し、マクロを 5

回呼び出してそれぞれを定義します。いずれも 2 つのオブジェクト (2 つ目は to_i を呼んで整数にする) から値を取り出し、マクロ引数で渡した式 exp により演算して結果を整数オブジェクトとして返します。6 つの比較演算については、値の真偽により予め用意してある 0 または 1 の整数オブジェクトへのポインタ値を返すところが違います。

```
void *int_to_i(void *p) {
  return p;
}
void *int_to_s(void *p) {
  char buf[100];
  sprintf(buf, "%d", ((ty_int*)p)->val); return str_new(buf);
}
void *int_print(void *p) {
  printf("%d", ((ty_int*)p)->val); return p;
}
void *int_println(void *p) {
  printf("%d\n", ((ty_int*)p)->val); return p;
}
#define icalc(name, exp) \
void *name(void *p, void *q) {\
  int x = ((ty_int*)p)->val, y = ((ty_int*)send0(q, "to_i"))->val; \
  return int_new(exp); }
icalc(int_add, x+y)
icalc(int_sub, x-y)
icalc(int_mul, x*y)
icalc(int_div, x/y)
icalc(int_mod, x%y)
#define icmp(name, cmp) \
void *name(void *p, void *q) {\
  int x = ((ty_int*)p)->val, y = ((ty_int*)send0(q, "to_i"))->val; \
  return (cmp) ?  &int_1_val : &int_0_val; }
icmp(int_gt, x>y)
icmp(int_lt, x<y)
icmp(int_ge, x>=y)
icmp(int_le, x<=y)
icmp(int_eq, x==y)
icmp(int_ne, x!=y)
char *int_names[] = {"to_i", "to_s", "print", "println", "add", "sub",
  "mul", "div", "mod", "gt", "lt", "ge", "le", "eq", "ne"};
int int_nargs[] = {0,0,0,0,1,1,1,1,1,1,1,1,1,1,1};
void* (*int_funcs[])() = {int_to_i, int_to_s, int_print, int_println,
  int_add, int_sub, int_mul, int_div, int_mod, int_gt, int_lt,
  int_ge, int_le, int_eq, int_ne};
void *int_new(int v) {
  ty_int *p = (ty_int*)malloc(sizeof(ty_int));
  p->tag = TAG_INT; p->nmethods = sizeof(int_nargs)/sizeof(int);
  p->funcs = int_funcs; p->names = int_names; p->nargs = int_nargs;
```

```
  p->val = v;
  return (void*)p;
}
ty_int int_1_val = {
  TAG_INT, sizeof(int_nargs)/sizeof(int), int_funcs, int_names, int_nargs, 1
};
ty_int int_0_val = {
  TAG_INT, sizeof(int_nargs)/sizeof(int), int_funcs, int_names, int_nargs, 0
};
```

メソッドテーブルはこれまでと同じです (だいぶメソッド数が多くなりましたが)。そして int_new も文字列のときと同じです。そのあと、前述の 1 と 0 のオブジェクトがあります。

では、これらの値を取り扱ってメソッドを起動する C プログラムの側を見てみましょう。まず配列 names の中を指定されたメソッド名と一致するものを探します (見つからなければエラー終了)。見つかったら配列 funcs の対応位置にアドレスが入っている関数を呼び出します。send0、send1 とも最初のパラメタはオブジェクト自身で、send1 はさらにもう 1 つパラメタを渡します。今回は簡単のため、渡せるパラメタの個数は 0 か 1 にしたので、これで十分です。

```
void *send0(void *p1, char *name) {
  ty_base *p = (ty_base*)p1;
  for(int i = 0; i < p->nmethods; ++i) {
    if(strcmp(p->names[i], name) == 0) {
      if(p->nargs[i] == 0) { return (p->funcs[i])(p1); }
      fprintf(stderr, "tag %d, name %s, wrong args: 0\n", p->tag, name);
      exit(1);
    }
  }
  fprintf(stderr, "tag %d, name %s, undefined method\n", p->tag, name);
  exit(1);
}
void *send1(void *p1, char *name, void *q1) {
  ty_base *p = (ty_base*)p1;
  for(int i = 0; i < p->nmethods; ++i) {
    if(strcmp(p->names[i], name) == 0) {
      if(p->nargs[i] == 1) { return (p->funcs[i])(p1, q1); }
      fprintf(stderr, "tag %d, name %s, wrong args: 1\n", p->tag, name);
      exit(1);
    }
  }
  fprintf(stderr, "tag %d, name %s, undefined method\n", p->tag, name);
  exit(1);
}
int tst(void *p1) {
  ty_base *p = (ty_base*)p1;
  if(p->tag == TAG_NIL) { return 0; }
  if(p->tag == TAG_INT) { return ((ty_int*)p)->val; }
  return 1;
```

```
}
```

最後の tst は、渡されたオブジェクトが nil なら偽、整数なら 0 が偽、それ以外は真、他のオブジェクトはすべて真として扱い、0 か 1 を返します。

では、これらのライブラリを呼び出してプログラムを実行する C 言語のプログラムを作ってみます。「整数を入力」をプロンプトとして用意し、文字列を入力して整数に変換し n に入れます。続いて i は n - 1 とし、while ループで n > 1 である間繰り返し、もし n % i == 0 なら「素数でない」と出力して終わります。そうでなければ i を 1 減らします。ループを抜けて来たら「素数です」と表示して終わります。

```
// test1.c --- demonstration for objlib usage.
#include "objlib.h"

int main(void) {
  void* s = str_new("input an integer> ");
  void* n = send0(send0(s, "prompt"), "to_i");
  void* i = send1(n, "sub", &int_1_val);
  while(tst(send1(i, "gt", &int_1_val))) {
    if(tst(send1(send1(n, "mod", i), "eq", &int_0_val))) {
      send0(str_new("not a prime number."), "println");
      return 0;
    }
    i = send1(i, "sub", &int_1_val);
  }
  send0(str_new("a prime number."), "println");
  return 0;
}
```

このように、値の操作はすべてオブジェクトと send$N$ で構成していますが、制御構造は C の制御構造を使います。実行のようすは次のようになります。

```
% gcc test3.c objlib.c
% ./a.out
input an integer> 97
a prime number.
% ./a.out
input an integer> 99
not a prime number.
%
```

演習 1 例題をそのまま動かせ。動いたら (ライブラリのオブジェクト群を使用することは前提として) 次のことをしてみよ。

a. 上の例題で使っていないメソッドを使った例題を作って動かせ。

b. 例題とは異なる枝分かれやループを含むプログラムを作ってみよ。

c. 自分でもクラスを何か 1 つ定義してみよ (大変)。

d. 配列がないと不便なので、配列に相当するクラスを作ってみよ。パラメタが 1 個なので「put は最後に get した位置に格納する」とかして 1 個で済ますか、send2 を実装してパラメタの最大を 2 個に増やすなどする (とても大変)。

3 小さなオブジェクト指向言語

3.1 文法記述

それでは、上のライブラリを使って小さなオブジェクトを実現してみます。コンパイラの出力はC言語のコードで、obj.cというファイルに出力されます。これと先のライブラリを一緒にコンパイルすると動くプログラムになります。

では文法記述から。演算等はすべてメソッド呼び出しなので演算子は言語に含まれていません (ただし後の演習問題を参照)。前述のように簡単にするため、メソッドのパラメタは0個か1個だけにしています。

```
Package samd1;

Helpers
  digit = ['0'..'9'] ;
  lcase = ['a'..'z'] ;
  ucase = ['A'..'Z'] ;
  letter = lcase | ucase | '_' ;
  graph = [' '..'~'] ;
  nodq = [graph - '"'] ;

Tokens
  sconst = '"' (nodq | '\' graph)* '"' ;
  iconst = digit+ ;
  blank = (' '|13|10)+ ;
  klass = 'class' ;
  method = 'method' ;
  end = 'end' ;
  var = 'var' ;
  nil = 'nil' ;
  new = 'new' ;
  if = 'if' ;
  while = 'while' ;
  semi = ';' ;
  comma = ',' ;
  dot = '.' ;
  assign = '=' ;
  lbra = '{' ;
  rbra = '}' ;
  lpar = '(' ;
  rpar = ')' ;
  ident = letter (letter|digit)* ;

Ignored Tokens
  blank;

Productions
```

```
prog = {class} cls prog
     | {empty}
     ;
cls  = {one} klass ident vars meths end
     ;
vars = {one} ident vars
     | {empty}
     ;
meths = {one} meth meths
      | {empty}
      ;
meth = {para} method ident lpar [para]:ident rpar lbra stlist rbra
     | {none} method ident lbra stlist rbra
     ;
stlist = {stat} stlist stat
       | {empty}
       ;
stat = {assign}  ident assign expr semi
     | {expr}    expr semi
     | {if}      if lpar expr rpar stat
     | {while}   while lpar expr rpar stat
     | {block}   lbra stlist rbra
     ;
expr = {fact} fact
     | {send0} expr dot ident
     | {send1} expr dot ident lpar [arg]:expr rpar
     ;
fact = {iconst} iconst
     | {sconst} sconst
     | {nil}    nil
     | {new}   new ident
     | {ident}  ident
     | {one}    lpar expr rpar
     ;
```

3.2 コンパイラドライバ

次はコンパイラのメインを見ます。動かし方ですが、コンパイラにはファイル名に加えて「最初に動かし始めるべきクラスとメソッド」を指定します。

記号表はこれまでとはまったく違う機能なので ObjSymtab というクラスを作りました。構文解析が終わったら、記号表を作り、まずクラス群をすべて登録します。これは ObjChecker というビジタークラスで行ないます。登録後に記号表の概要を表示しています。

続いて、各メソッドの中の名前参照をチェックします。パスが分けてあるのは、ソースプログラム上で先の方にあるクラスでも参照できるようにするためなのでした。この処理は VarChecker というビジタークラスで行ないます。ここまででエラーがあればコード生成せずに終わります。

エラーが無ければ、C 言語ヘッダファイル obj.h と C 言語ソースファイル obj.c を生成する準備をします。ヘッダファイル側は記号表のメソッド emithdr を呼ぶことで必要なヘッダ用定義を出力し

ます。

```
package samd1;
import samd1.parser.*;
import samd1.lexer.*;
import samd1.node.*;
import java.io.*;
import java.util.*;

public class SamD1 {
  public static void main(String[] args) throws Exception {
    if(args.length != 3) {
      Log.pError("usage: SamD1 srcfile startclass startmethod.");
      System.exit(1);
    }
    Parser p = new Parser(new Lexer(new PushbackReader(
      new InputStreamReader(new FileInputStream(args[0]), "JISAutoDetect"),
        1024)));
    Start tree = p.parse();
    ObjSymtab st = new ObjSymtab();
    ObjChecker ck = new ObjChecker(st); tree.apply(ck); st.show();
    VarChecker vc = new VarChecker(st); tree.apply(vc);
    if(Log.getError() > 0) { return; }
    PrintStream hp = new PrintStream("obj.h"); st.emithdr(hp); hp.close();
    PrintStream pr = new PrintStream("obj.c");
    pr.println("#include <stdlib.h>");
    pr.println("#include \"objlib.h\"");
    pr.println("#include \"obj.h\"");
    Generator gen = new Generator(st, pr); tree.apply(gen);
    st.emitbody(pr, args[1], args[2]); pr.close();
  }
}
```

クラス Log は変更していませんが一応掲載します。

```
package samd1;
public class Log {
  public static int err = 0;
  public static void pError(String s) { System.out.println(s); ++err; }
  public static int getError() { return err; }
}
```

3.3 記号表

次は記号表ですが、今度の言語は「クラスとその中のメソッド」という 2 段階になっているのでこれまでとは全く構造が違い、作り直しています (ただし、非常に簡素化してあります)。

定数と変数から説明しましょう。まず、クラスごとに固有の ID を持たせますが、その最初は 4 からとします。次に、変数の種類としては「インスタンス変数」「メソッドの結果用変数」「ローカル変数」の 3 種類を扱います。ただし簡単のため、ローカル変数はパラメタだけで、通常の変数はすべて

インスタンス変数です。そして、メソッド名と同じ名前の変数が 1 つ用意されていて、返値を設定するにはこの変数に代入します。これがメソッドの結果変数です。

クラスに関する情報は内部のクラス ClassData が扱うので、主要なデータ構造は名前文字列から ClassData オブジェクトを検索できる表 cmap です。そして、現在処理中のクラスを変数 cur に保持します。また、現在のメソッドのパラメタ名実行を開始するクラスとそのメソッドのパラメタ名を pnm、メソッド名を mnm に保持します。

```
package samd1;
import java.util.*;
import java.io.*;

public class ObjSymtab {
  public static int newid = 4;
  public static int IVAR = 1, MVAR = 2, LVAR = 3, UNDEF = 0;
  public Map<String,ClassData> cmap = new TreeMap<String,ClassData>();
  public ClassData cur = null;
  String pnm = null, mnm = null;
  public void enterClass(String n) {
    if(!cmap.containsKey(n)) { cmap.put(n, new ClassData(n)); }
    cur = cmap.get(n);
  }
  public void addIvar(String n) {
    if(!cur.vset.contains(n)) { cur.vset.add(n); return; }
    Log.pError(cur.name + ": dupl var " + n);
  }
  public void addMethod(String m, String p) {
    if(cur.mmap.containsKey(m)) { Log.pError(cur.name+": dupl method "+m); }
    else { cur.mmap.put(m, p); }
  }
  public void addMethod(String m) {
    if(cur.mmap.containsKey(m)) { Log.pError(cur.name+": dupl method "+m); }
    else { cur.mmap.put(m, null); }
  }
  public void enterMethod(String m, String p) { mnm = m; pnm = p; }
  public void enterMethod(String m) { mnm = m; pnm = null; }
  public int vkind(String n) {
    if(pnm != null && pnm.equals(n)) { return LVAR; }
    if(mnm != null && mnm.equals(n)) { return MVAR; }
    if(cur.vset.contains(n)) { return IVAR; }
    return UNDEF;
  }
  public boolean isClass(String n) { return cmap.containsKey(n); }
  public void show() {
    for(String n: cmap.keySet()) { cmap.get(n).show(); }
  }
  public void emithdr(PrintStream pr) {
    for(String n: cmap.keySet()) { cmap.get(n).emithdr(pr); }
```

```
  }
  public void emitbody(PrintStream pr, String scls, String smeth) {
    ClassData c = cmap.get(scls);
    if(c == null || !c.mmap.containsKey(smeth) || c.mmap.get(smeth) != null) {
      Log.pError("invalid start class "+scls+" or method "+smeth); return;
    }
    for(String n: cmap.keySet()) { cmap.get(n).emitbody(pr); }
    pr.println("int main(void) {");
    pr.printf(" send0(%s_new(), \"%s\"); return 0; }\n", scls, smeth);
  }
```

enterClass() はクラスに入るときに呼ばれ、そのクラスの ClassData が無ければインスタンスを生成して cmap に登録し、cur にそのクラスの ClassData オブジェクトを保持さます。

addIvar() はインスタンス変数を追加します。実際には classData オブジェクトが保持するデータ構造に登録するだけです。2 重定義ならエラーとします。

addMethod() はメソッドを追加しますが、これも ClassData オブジェクトが保持するデータ構造に追加します。enterMethod() は 2 パス目以降でメソッドに入る時に呼び、パラメタ文字列 (または null)、メソッド名を pnm、mnm に保持します。

isClass() は渡された文字列がクラス名として登録されているか否かを調べて返します。show()、emithdr() はいずれも各クラスについて show()、emithdr() を呼ぶだけです。emitbody() は実行開始クラス/メソッド名が正しいかチェックしたあと、各クラスについて emitbody() を呼び、最後に「開始クラスのインスタンスを生成して開始メソッドを呼ぶ」だけの C 言語の main() を生成します。

ClassData は基本的にそのクラスのメソッド群を保持する mmap とインスタンス変数群を保持する vset を持つことが主な役割です。これらのインスタンス変数は直接外から (といっても ObjSymtab の中だけですが) 参照できます。

show() はこのクラスの名前、インスタンス変数のリスト、そして各メソッドの名前と (あれば) パラメタを順次表示します。emithdr() はこのクラスに対応する構造体定義を出力します。ほとんどはどのクラスでも同じで、クラス名の部分とインスタンス変数のところが違うだけです。

```
  static class ClassData {
    public String name;
    public int id = newid++;
    public Map<String,String> mmap = new TreeMap<String,String>();
    public Set<String> vset = new TreeSet<String>();
    public ClassData(String n) { name = n; }
    public void show() {
      System.out.println(name + ":");
      for(String v: vset) { System.out.print(" " + v); }
      System.out.println();
      for(String k: mmap.keySet()) {
        String v = mmap.get(k);
        if(v == null) { System.out.printf(" %s()\n", k); }
        else          { System.out.printf(" %s(%s)\n", k, v); }
      }
    }
    public void emithdr(PrintStream pr) {
      pr.println("typedef struct {");
```

```
      pr.println(" int tag, nmethods; void* (**funcs)();");
      pr.println(" char* *names; int *nargs;");
      for(String v: vset) { pr.println(" void *_"+v+";"); }
      pr.println("} ty_"+name+";");
      pr.println("void *"+name+"_new();");
    }
    public void emitbody(PrintStream pr) {
      String n = name;
      pr.printf("char *%s_names[] = {", n);
      for(String m: mmap.keySet()) { pr.printf("\"%s\",", m); }
      pr.println("};");
      pr.printf("int %s_nargs[] = {", n);
      for(String m: mmap.keySet()) { pr.print(mmap.get(m) == null ? "0,":"1,"); }
      pr.println("};");
      pr.printf("void* (*%s_funcs[])() = {", n);
      for(String m: mmap.keySet()) { pr.printf("%s_%s,", n, m); }
      pr.println("};");
      pr.printf("void *%s_new() {\n", n);
      pr.printf(" ty_%s *p = (ty_%s*)malloc(sizeof(ty_%s));\n", n, n, n);
      pr.printf(" p->tag=%d; p->nmethods=sizeof(%s_names)/sizeof(int);\n", id, n);
      pr.printf(" p->funcs=%s_funcs; p->names=%s_names; p->nargs=%s_nargs;\n",
                n, n, n);
      for(String v: vset) { pr.printf(" p->_%s=&nil_val;\n", v); }
      pr.println(" return (void*)p; }");
    }
  }
} // ObjSymtab の終わり
```

emitbody() はまず前半でメソッドテーブルの 3 つの配列を出力します。そして後半でこのクラスのインスタンスを生成するメソッドの定義を出力します。構造はどのクラスでもほとんど同じですが、インスタンス変数群をすべて nil に初期化するところはクラスごとに違っています。

3.4 意味解析

意味解析はそれ自体が 2 パスになっています (クラスについて前方参照したいため)。1 パス目を実行するのが ObjChecker で、クラスの入口でクラスオブジェクトを用意し、インスタンス変数宣言ごとに記号表にインスタンス変数を追加し、またメソッドの入口でメソッド名と (あれば) パラメタ名を登録します。

```
package samd1;
import samd1.analysis.*;
import samd1.node.*;

public class ObjChecker extends DepthFirstAdapter {
  ObjSymtab st;
  public ObjChecker(ObjSymtab st1) { st = st1; }
  @Override
  public void inAOneCls(AOneCls node) {
```

```
    st.enterClass(node.getIdent().getText());
  }
  @Override
  public void outAOneVars(AOneVars node) {
    st.addIvar(node.getIdent().getText());
  }
  @Override
  public void inAParaMeth(AParaMeth node) {
    st.addMethod(node.getIdent().getText(), node.getPara().getText());
  }
  @Override
  public void inANoneMeth(ANoneMeth node) {
    st.addMethod(node.getIdent().getText());
  }
}
```

2パス目はVarCheckerが実行しますが、クラスの入口やメソッドの入口では記号表をそのクラス/メソッドの状態に設定するためにenterClass()、enterMethod()を呼びます。実際のチェックは代入文のところで左辺の変数があることを確認すること、変数名が現れたときにその変数があることを確認すること、そして「new クラス名」が現れたときにそのクラスが存在することを確認することです。

```
package samd1;
import samd1.analysis.*;
import samd1.node.*;

public class VarChecker extends DepthFirstAdapter {
  ObjSymtab st;
  public VarChecker(ObjSymtab st1) { st = st1; }
  @Override
  public void inAOneCls(AOneCls node) {
    st.enterClass(node.getIdent().getText());
  }
  @Override
  public void inAParaMeth(AParaMeth node) {
    st.enterMethod(node.getIdent().getText(), node.getPara().getText());
  }
  @Override
  public void inANoneMeth(ANoneMeth node) {
    st.enterMethod(node.getIdent().getText());
  }
  @Override
  public void outAAssignStat(AAssignStat node) {
    if(st.vkind(node.getIdent().getText()) != ObjSymtab.UNDEF) { return; }
    Log.pError("undefined var: " + node.getIdent().getText());
  }
  @Override
```

```
  public void outAIdentFact(AIdentFact node) {
    if(st.vkind(node.getIdent().getText()) != ObjSymtab.UNDEF) { return; }
    Log.pError("undefined var: " + node.getIdent().getText());
  }
  @Override
  public void outANewFact(ANewFact node) {
    if(st.isClass(node.getIdent().getText())) { return; }
    Log.pError("undefined class in new: " + node.getIdent().getText());
  }
}
```

3.5 コード生成

では最後にコード生成です。方針として、式はその式に対応する C 言語の式のコードを文字列として構成し、文から上では 1 行ずつコードを出力しています。まずメソッドの入口ではメソッドの定義とオブジェクトの型へのキャストを生成し、結果変数を初期化します。出口では結果変数を値として return します。

代入文は変数の種類によって変数名そのまま、または「`self->_`」を先頭につけた形にし、右辺は式の文字列をそのまま埋め込んで C の代入を生成します。if や while は入口で条件部をまず文字列として取得し、それを用いて C の if/while の先頭を生成してから、本体部分を生成し、最後に「`}`」を生成します。

```
package samd1;
import samd1.analysis.*;
import samd1.node.*;
import java.io.*;

public class Generator extends DepthFirstAdapter {
  ObjSymtab st;
  PrintStream pr;
  String cname, mname, pname;
  public Generator(ObjSymtab s, PrintStream p) { st = s; pr = p; }
  @Override
  public void inAOneCls(AOneCls node) {
    cname = node.getIdent().getText(); st.enterClass(cname);
  }
  @Override
  public void inAParaMeth(AParaMeth node) {
    pname = node.getPara().getText(); mname = node.getIdent().getText();
    pr.printf("void *%s_%s(void *_self, void *%s) {\n", cname, mname, pname);
    pr.printf(" ty_%s *self = (ty_%s*)_self;\n", cname, cname);
    pr.printf(" void *%s = &nil_val;\n", mname);
  }
  @Override
  public void outAParaMeth(AParaMeth node) {
    pr.printf(" return %s; }\n", mname);
  }
  @Override
```

```
public void inANoneMeth(ANoneMeth node) {
  pname = ""; mname = node.getIdent().getText();
  pr.printf("void *%s_%s(void *_self) {\n", cname, mname);
  pr.printf(" ty_%s *self = (ty_%s*)_self;\n", cname, cname);
  pr.printf(" void *%s = &nil_val;\n", mname);
}
@Override
public void outANoneMeth(ANoneMeth node) {
  pr.printf(" return %s; }\n", mname);
}
@Override
public void outAAssignStat(AAssignStat node) {
  String v = node.getIdent().getText(), e = (String)getOut(node.getExpr());
  if(st.vkind(v) == ObjSymtab.IVAR) { v = "self->_" + v; }
  pr.printf(" %s = %s;\n", v, e);
}
@Override
public void outAExprStat(AExprStat node) {
  pr.printf(" %s;\n", ((String)getOut(node.getExpr())));
}
@Override
public void caseAIfStat(AIfStat node) {
  node.getExpr().apply(this); String e = (String)getOut(node.getExpr());
  pr.printf(" if(tst(%e)) {\n", e);
  node.getStat().apply(this);
  pr.println(" }");
}
@Override
public void caseAWhileStat(AWhileStat node) {
  node.getExpr().apply(this); String e = (String)getOut(node.getExpr());
  pr.printf(" while(tst(%s)) {\n", e);
  node.getStat().apply(this);
  pr.println(" }");
}
@Override
public void outAFactExpr(AFactExpr node) {
  setOut(node, getOut(node.getFact()));
}
@Override
public void outASend0Expr(ASend0Expr node) {
  setOut(node, String.format("send0(%s, \"%s\")",
          (String)getOut(node.getExpr()), node.getIdent().getText()));
}
@Override
public void outASend1Expr(ASend1Expr node) {
  setOut(node, String.format("send1(%s, \"%s\", %s)",
```

```
            (String)getOut(node.getExpr()), node.getIdent().getText(),
            (String)getOut(node.getArg())));
  }
  @Override
  public void outAIconstFact(AIconstFact node) {
    setOut(node, String.format("int_new(%s)", node.getIconst().getText()));
  }
  @Override
  public void outASconstFact(ASconstFact node) {
    setOut(node, String.format("str_new(%s)", node.getSconst().getText()));
  }
  @Override
  public void outANilFact(ANilFact node) {
    setOut(node, "&nil_val");
  }
  @Override
  public void outANewFact(ANewFact node) {
    setOut(node, String.format("%s_new()", node.getIdent().getText()));
  }
  @Override
  public void outAIdentFact(AIdentFact node) {
    String v = node.getIdent().getText();
    if(st.vkind(v) == ObjSymtab.IVAR) { v = "self->_" + v; }
    setOut(node, v);
  }
  @Override
  public void outAOneFact(AOneFact node) {
    setOut(node, getOut(node.getExpr()));
  }
}
```

式から先は send$N$(...) やオブジェクトの生成を C コードの文字列として構築していくだけです。

3.6 実行例と生成コード

では小さなオブジェクト指向言語による例題コードを示します。まず、myclass というクラスは put された値を次々に加算し、get で現在の累計が取得できるというものです。そして main クラスの start で実行を開始しますが、これはプロンプトを出して 0 が入力されるまで次々に上記の put を呼んで値を累計し、最後に合計を出力します。

```
class main
  acc v
  method start {
    acc = new myclass;
    v = "enter number or '0'> ".prompt.to_i;
    while(v.ne(0)) {
      acc.put(v);
      v = "enter number or '0'> ".prompt.to_i;
```

```
    }
    "total = ".add(acc.get).println;
  }
end
class myclass
  mem
  method put(v) { mem = v.add(mem); }
  method get { get = mem; }
end
```

動かす様子を示します。

```
% java samd1/SamD1 test.obj main start
main:
 acc v
 start()
myclass:
 mem
 get()
 put(v)
% gcc obj.c objlib.c
% ./a.out
enter number or '0'> 3
enter number or '0'> 5
enter number or '0'> 7
enter number or '0'> 0
total = 15
%
```

参考までに生成されたコードを見てみましょう。まずヘッダですが、構造体の定義と型名の定義、そして生成メソッドのプロトタイプ宣言をクラスごとに出力します。構造体の中身はタグとメソッド表までは同じですが、インスタンス変数についてはクラス毎に違います。

```
typedef struct {
 int tag, nmethods; void* (**funcs)();
 char* *names; int *nargs;
 void *_acc;
 void *_v;
} ty_main;
void *main_new();
typedef struct {
 int tag, nmethods; void* (**funcs)();
 char* *names; int *nargs;
 void *_mem;
} ty_myclass;
void *myclass_new();
```

次にコードですが、まず各クラスのインスタンスメソッドがすべて出力され、その後でクラス毎にメソッドテーブルの配列定義とオブジェクトを生成する関数が生成されます。

```
#include <stdlib.h>
#include "objlib.h"
#include "obj.h"
void *main_start(void *_self) {
 ty_main *self = (ty_main*)_self;
 void *start = &nil_val;
 self->_acc = myclass_new();
 self->_v = send0(send0(str_new("enter number or '0'> "), "prompt"), "to_i");
 while(tst(send1(self->_v, "ne", int_new(0)))) {
 send1(self->_acc, "put", self->_v);
 self->_v = send0(send0(str_new("enter number or '0'> "), "prompt"), "to_i");
 }
 send0(send1(str_new("total = "), "add", send0(self->_acc, "get")), "println");
 return start; }
void *myclass_put(void *_self, void *v) {
 ty_myclass *self = (ty_myclass*)_self;
 void *put = &nil_val;
 self->_mem = send1(v, "add", self->_mem);
 return put; }
void *myclass_get(void *_self) {
 ty_myclass *self = (ty_myclass*)_self;
 void *get = &nil_val;
 get = self->_mem;
 return get; }
char *main_names[] = {"start",};
int main_nargs[] = {0,};
void* (*main_funcs[])() = {main_start,};
void *main_new() {
 ty_main *p = (ty_main*)malloc(sizeof(ty_main));
 p->tag=4; p->nmethods=sizeof(main_names)/sizeof(int);
 p->funcs=main_funcs; p->names=main_names; p->nargs=main_nargs;
 p->_acc=&nil_val;
 p->_v=&nil_val;
 return (void*)p; }
char *myclass_names[] = {"get","put",};
int myclass_nargs[] = {0,1,};
void* (*myclass_funcs[])() = {myclass_get,myclass_put,};
void *myclass_new() {
 ty_myclass *p = (ty_myclass*)malloc(sizeof(ty_myclass));
 p->tag=5; p->nmethods=sizeof(myclass_names)/sizeof(int);
 p->funcs=myclass_funcs; p->names=myclass_names; p->nargs=myclass_nargs;
 p->_mem=&nil_val;
 return (void*)p; }
int main(void) {
 send0(main_new(), "start"); return 0; }
```

演習 2 「小さなオブジェクト指向言語」を次のように拡張してみよ。

a. 現在の言語では「`1 + 2`」のような中置記法は使えず、「`1.add(2)`」のようにメッセージ送信記法で書く必要がある。これでは見にくいので、演算子が扱えるように構文を変更してみよ (ヒント: 構文上は中置記法であっても、生成される C 言語コードはこれまでと同様に send$N$ を呼べばよい)。
b. 現在の言語ではそれぞれのクラスは独立している。これを拡張して、継承機構を追加してみよ。クラスに親クラスが指定されている場合は、インスタンス変数とメソッドをその親クラスから取り込んでくる、というのが想定される継承の機能である。
c. その他自分の好きなように言語を拡張してみよ。

4 型検査と静的なメソッドテーブル

4.1 型のあるオブジェクト指向言語

ここまで見て来た「小さなオブジェクト指向言語」の実装では、メソッドテーブルは「メソッドが定義された順」で並んでいて、実行時にメソッド名 (文字列) をキーとして線形探索で実際に呼ぶメソッドを決定していました。しかしこれでは高速に実行できないのは明らかです。

また、クラス名の未定義は検出されるが、メソッド名の未定義は実行時に呼び出すまで分かりません。これは型検査を行なわないため、どうにもなりません。

Smalltalk-80 など初期のオブジェクト指向言語ではこれらを言語仕様として受け入れていましたが、オブジェクト指向が普及し始めた時期 (1980 年代) に、これらの問題を解消するためにオブジェクト指向言語に型検査を導入しようと、多くの研究がなされました。今日の C++言語や Java 言語はそれらの研究の成果を取り入れて設計されています。ここでは Java を例として説明しましょう。

Java 言語ではすべてのオブジェクトは対応するクラスが型となっています。たとえば `MyClass` のインスタンスであれば型も `MyClass` です。

そして次に、`Object` を除くすべてのクラスは 1 つだけ親クラスを持ちます。クラス定義の冒頭において `extends` キーワードで親クラスを指定しますが、その指定がない場合は `Object` が指定されたものとして扱います。従って、すべてのクラスは直接または間接に `Object` の子孫となります。

ここで型検査の規則は次のようになります。

> クラス T, T' において、T が T' の子孫であることを $T < T'$ と表す。変数の型が T' のとき、その変数に代入できる値は $T \leq T'$ なる T 型の値に限られる。

この規則の「根拠」ですが、継承関係にある場合、子クラスは親クラスからメソッドを引き継ぐため、親クラス T' が持つメソッドの情報に基づいて型検査を行なった場合、子クラス T のオブジェクトは同じメソッド群を持つので正しく実行できる、というところにあります。

また、メソッドを親クラスからコピーしてきてそのまま使えるかという点については、子クラスのインスタンスは親クラスのインスタンス変数をすべて持ち、さらに独自のインスタンス変数を追加する、という設計であるため、配置を工夫すれば問題なく扱えます。この点は次節で具体的に説明します。

4.2 単一継承向けのインスタンスとメソッドテーブル設計

具体例があった方が分かりやすいので、「図形クラスと、そのサブクラスとしての円と長方形」という例を示します。図形クラスは抽象クラスで、両方の図形に共通のインスタンス変数・メソッド群を持ちます。そして円と長方形は図形の子クラスで、それぞれ独自のインスタンス変数とメソッドを追加しています。また、メソッド `draw()` は抽象メソッドで、図形クラスでは宣言のみされていて、円と長方形で具体的な定義がなされています。

```
abstract class Figure {
  int xpos, ypos;
  Color col;
  public Figure(int x, int y, Color c) { ... 初期化 ... }
  public void moveTo(int x, int y) { xpos = x; ypos = y; }
  public void setColor(Color c) { col = c; }
  public abstract void draw(Graphics g);
}
class Circle extends Figure {
  int rad;
  public Circle(int x, int y, int r) { ... 初期化 ... }
  public void draw(Graphics g) { ... 円の描画 ... }
  public void scale(float r) { rad = (int)(rad * r); }
}
class Rect extends Figure {
  int with, height;
  public Rect(int x, int y, int w, int h) { ... 初期化 ... }
  public void draw(Graphics g) { ... 長方形の描画 ... }
  public void rot90() { int z = w; w = h; h = z; }
}
```

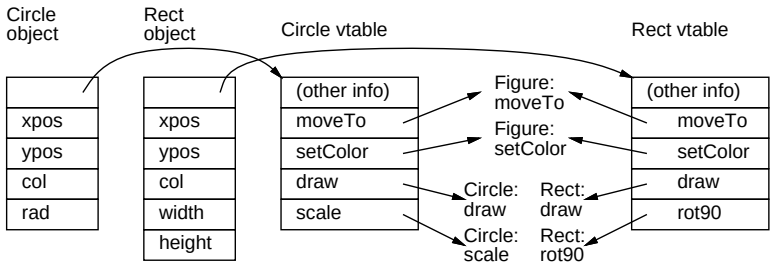

図 3: 単一継承むけの静的な配置

このような単一継承の体系では、図 3 のように、どの子クラスのオブジェクトも、親クラスの部分の「後ろに」自分独自のインスタンス変数の領域を確保する配置とすることで、親クラスの部分はすべて共通に保つことができます。

そして、各オブジェクトは自クラスのメソッドテーブルへのポインタを持つようにしますが、それぞれのクラスのメソッドテーブルもインスタンスと同様、親クラスのメソッド群をまず配置し、その後ろに独自のメソッドのスロットを配置します。

このようにしておけば、たとえば Figure:moveTo は Circle に対しても Rect に対しても呼び出されますが、インスタンス変数としてアクセスする部分は Figure で定義されているものだけなので、後ろの方が違っていても感知しません。また、draw はどちらのクラスのメソッドテーブルでも 3 スロット目と共通の位置にありますが、実際に呼ばれるものは Circle と Rect で別のものです。そしてその下には、それぞれのクラス独自のメソッドが配置されます。

これにより、インスタンス変数のアクセスもメソッドの呼び出しもオブジェクトやメソッドテーブルの先頭から固定オフセットにコンパイルでき、効率のよい実行が行なえます。

この方法は、親クラスが 1 つだけという「単一継承」の設計だから可能になるものであり、親が複数ある多重継承ではうまく行きません。多重継承の場合は、実行時に探索を行なうか、もっと複雑な構造を取り入れる必要がありますが、ここでは触れません。

演習 3 静的な型検査とここで説明したメソッドテーブルを持つオブジェクト指向言語を実装してみなさい。

5 課題 13A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題# 13」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. 強い型の言語と弱い型の言語のどちらが好みですか。またそれはなぜ。
  Q2. 記号表と型検査の実装について学んでみて、どのように思いましたか。
  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。