プログラミング言語設計論

2014年度

第2週: 抽象データ型

担当: 増原英彦

おさらい

Parnasモジュール化原則 [Par72]

... begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.

問題1 (10分)

右は文字列中で() [ ] { }という三種類の括弧対応の正しさを判定するモジュール性の観点から批判せよ

アルゴリズム: 左から順に、開き括弧が来たらそれをスタックに積む。閉じ括弧が来たら、スタックから取り出した開き括弧との対応を調べる

```
int corresponding(char* string) {
  char* stack = malloc(...);
  int top = 0;
  for (int i = 0; i < strlen(string); i++) {
    char c = string[i];
    if (is_open(c))
     stack[top++] = c;
    else if (is_close(c) && (top==0 ||
                    stack[--top] != opening(c))
        return false;
  }
  return true;
}
```

開き括弧か調べる

閉じ括弧か調べる

対応する開き括弧を得る

問題2 (10分)

スタックを分離した。
モジュール性の観点から批判せよ

```
typedef struct stack { char* c; int t; } stk;
extern stk* stk_new(int len);
extern void stk_push(stk* s, char c);
extern char stk_pop(stk* s);
extern int stk_empty(stk* s);
```

stack.h

```
#include "stack.h"
typedef struct stack { char* c; int t; } stk;
stk* stk_new(int len) { ... }
void stk_push(stk* s, char c)
{ s->c[s->t++] = c; }
char stk_pop(stk* s) { return s->c[--s->t]; }
int stk_empty(stk* s) { return s->t == 0; }
```

stack.c

```
#include "stack.h"
int corresponding(char* str) {
  stk* s = stack_new(...);
  int i;
  for (i = 0; i < ...; i++) {
    char c = str[i];
    if(is_open(c))
      stk_push(s,c);
    else if (is_close(c) &&
      (stk_empty(s) ||
      stk_pop(s) != opening(c)))
       return false;
  }
  return 1;
}
```

問題3 (5分)

- モジュールのインタフェースは何を規定するべきか考えよ
  - 注: モジュールはインタフェースと実現からなる
  - 注: 規定する = どんな情報を書くべきか
  - ➢スタックのようなデータ構造をモジュールとして定義する場合に限ってよい

データ構造の内部表現 (問題1)

- 変わりやすい決定 = データの内部表現
- 例: スタック
  - 配列 + 先頭の添字
  - リスト

```
int corresponding(char* string) {
  char* stack = malloc(...);
  int top = 0;
  for (int i = 0; i < strlen(string); i++) {
    char c = string[i];
    if (is_open(c))
     stack[top++] = c;
    else if (is_close(c) && (top==0 ||
                        stack[--top] != opening(c))
        return false;
  }
  return true;
}
```

モジュール機構: 一般論

使用者

インタフェースに規定された名前を通して実現を使用

インタフェース

関数名・変数名・型名を規定

実現

関数定義・変数定義・型の定義
(インタフェースに規定されたものを必ず含む)

C言語のモジュール機構

インタフェース: *h (ヘッダファイル)
extern付きプロトタイプ宣言

```
typedef struct stack { char* c; int t; } stk;
extern stk* stk_new(int len);
extern void stk_push(stk* s, char c);
extern char stk_pop(stk* s);
extern int stk_empty(stk* s);
```

stack.h

```
#include "stack.h"
int corresponding(char* str) {
  stk* s = stack_new(...);
  int i;
  for (i = 0; i < ...; i++) {
    char c = str[i];
    if (is_open(c)
      stk_push(s, c);
    else if (is_close(c) &&
      stk_pop(s) != opening(c))
      return false;
  }
  return 1;
}
```

使用者
#includeで指定
同名の関数・変数を使用

stack.c

```
#include "stack.h"
typedef struct stack { char* c; int t; } stk;
stk* stk_new(int len) { ... }
void stk_push(stk* s, char c)
{ s->c[stk->t++] = c; }
char stk_pop(stk* s) { return s
int stk_empty(stk* s) { return s
```

実現
#includeでインタフェースを指定
同名の関数・変数を定義する

モジュール機構の強制力

- 強制力あり(情報隠蔽): インタフェースに規定された名前のみしか使用できない
  - それ以外を使うとエラー
  - 提供者は、インタフェースにない定義を変更しても使用者に影響を与えないことを確信
- 強制力なし: インタフェースはガイドラインとして振舞う

C言語: 強制力なし

- 別のモジュールで定義された変数・関数は extern 宣言を書けば参照可能

stack.c

```
#include “stack.h”
int top = 0;
void stk_push(…) { …
```

```
extern void stk_push(…);
```

stack.h

```
#include “stack.h”
extern int top;
int main( … ) {
   … puts( top ) …
}
```

user.c

データ型をモジュール化する際の問題 (問題2)

使用者は内部表現に依存してしまう

```
#include "stack.h"
int corresponding(char* str) {
  stk* s = stack_new(...);
  int i;
  for (i = 0; i < ...; i++) {
    char c = str[i];
```

```
typedef struct stack { char* c; int t; } stk;
extern stk* stk_new(int len);
```

定義されるデータの内部表現を使用者から隠したい!

だけど

使用者は定義されたデータ型を使いたい!

抽象データ型[LZ74]

- データに特化したモジュール化機構

使用者

- 提供されるデータの「型」名を使う
- 提供される関数を使う

インタフェース

- 提供する型の名前
- 提供する関数の名前

実現

- 提供するデータ型の内部表現
- 内部表現に依存した関数の定義

CLU言語

[LZ74, LAB+79]

データ型の定義

データ型 =
提供する関数 +
内部表現 +
関数の実現

```
stack: cluster is push, pop, empty;
  rep =
    (tp: integer;
     stk: array[1..] of char;)
  create 略 end
  push: operation(s: rep, v: char);
    s.tp := s.tp+1;
    s.stk[s.tp] := v;
    return;
  end
  pop: operation(s: rep) returns char;
    s.tp := s.tp -1;
    return s.stk[s.tp+1];
  end
  empty: operation(s: rep) returns boolean;
    略
end stack
```

提供する関数

内部表現

関数の実現

CLU言語

[LZ74, LAB+79]

データ型の定義

```
stack: cluster is push, pop, empty;
  rep =
    (tp: integer;
     stk: array[1..] of char;)
  create 略 end
  push: operation(s: rep, v: char);
    s.tp := s.tp+1;
    s.stk[s.tp] := v;
    return;
  end
  pop: operation(s: rep) returns char;
    s.tp := s.tp -1;
    return s.stk[s.tp+1];
  end
  empty: operation(s: rep) returns boolean;
    略
end stack
```

提供する関数

内部表現

関数の実現

使用者

```
str: array[1..] of char;
s: stack;
i: integer;

for i = 1 to ...
    char c = str[i];
    if(is_open(c))
      stack$push(s,c);
    else if (is_close(c) &&
      stack$empty(s) ||
      stack$pop(s) != opening(c))
            return false;
end for;
return 1;
```

型の使用

モジュール化が与える保証

- モジュール化は
  - インタフェースを変えない
  - 提供者はインタフェースに規定された関数などを提供する
  - 使用者はインタフェースに定義された関数のみを使用する

限り、提供者が安全?に実現を変更できることを保証

モジュール機構の保証する安全性

- プログラムが実行可能になる
  vs. モジュールをつなげようとしたらつながらない!
- プログラムが「動く」
  vs. 関数を呼ぼうとしたら、定義されていなかった!
  vs. 関数を呼んだら止まった!

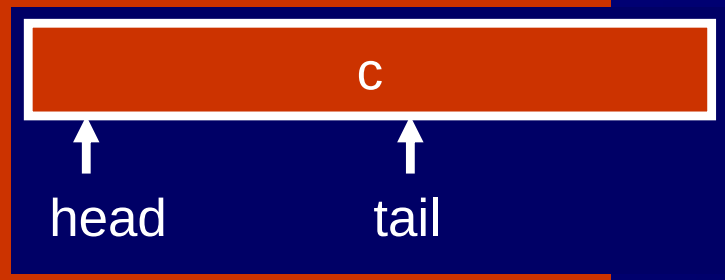
～型安全であることと同義

型安全性: プログラムが期待した通りの型の
データを受け渡しする

これは正しいスタックの定義か?

stack.c

```
#include "stack.h"
struct { char* c; int size; int head; int tail; } stk;
stk* stk_new(int len) { ... }
void stk_push(stk* s, char c) {
   s->c[stk->tail] = c;
   stk->tail = (stk->tail+1)%stk->size; }
char stk_pop(stk* s) {
  stk->head = (stk->head+1)%stk->size;
  return s->c[stk->head]; }
int stk_empty(stk* s) { return s->tail == s->head; }
```



Liskov置換原則 [Lis87]

If for each object o_1 of type S there is an object o_2 of type T such that for all program P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.

振舞部分型[LW94] (behavioral subtyping)

- インタフェースはモジュールの抽象状態と振舞を規定する
  - 振舞: 事前条件 + 事後条件
- 例: スタック
  - (抽象状態) l : 要素のリスト
  - push(x) : (事後条件) $l_{post} = l_{pre} :: x$
  - pop(x) : (事前条件) $l_{pre} \neq \phi$
    (事後条件) $ret = last(l_{pre}), l_{post} = butlast(l_{pre})$

振舞部分型の使い方

- インタフェース
  - 抽象状態と振舞
- 提供者
  - 内部状態と抽象状態の対応を決めて
  - 「事前条件を満たしているときに実行されたら、事後条件が満たされる」ように関数を作る
- 使用者

参考文献

- [LZ74] Barbara Liskov, Stephen Zilles, Programming with Abstract Data Types, *Proceedings of Symposium on Very High Level Languages*, 1974
- [LAB+79] Barbara Liskov, et al. *CLU Reference Manual*, 1979
- [Lis87] Barbara Liskov, Data Abstraction and Hierarchy, *Keynote Talk at OOPSLA'87*, 1987.
- [LW94] Barbara Liskov and Jeannette Wing, A behavioral notion of subtyping, *Transactions on Programming Languages and Systems*, Vol.16, No.6, pp.1811-1841, November 1994.