Programming Language Design

2015

Week #2: Abstract datatypes

Instructor: Hidehiko Masuhara

Review

Modularization Principle of Parnas [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.

Quiz #1 (10min)

Criticize the code from a viewpoint of modularity.

It determines whether parentheses like
() [ ] { } in a string match.

Algorithm: scan from left to right; with an open paren. push it on the stack; with a close paren, pop from the stack and check matching.

```
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;
}
```

is c an open paren?

is c a close paren?

gives a matching open paren for c

Quiz #2 (10min.)

We separate the implementation of the stack. Criticize from a viewpoint of modularity.

stack.h

```
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.c

```
#include "stack.h"

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; }
```

```
#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;
}
```

Quiz #3 (5min.)

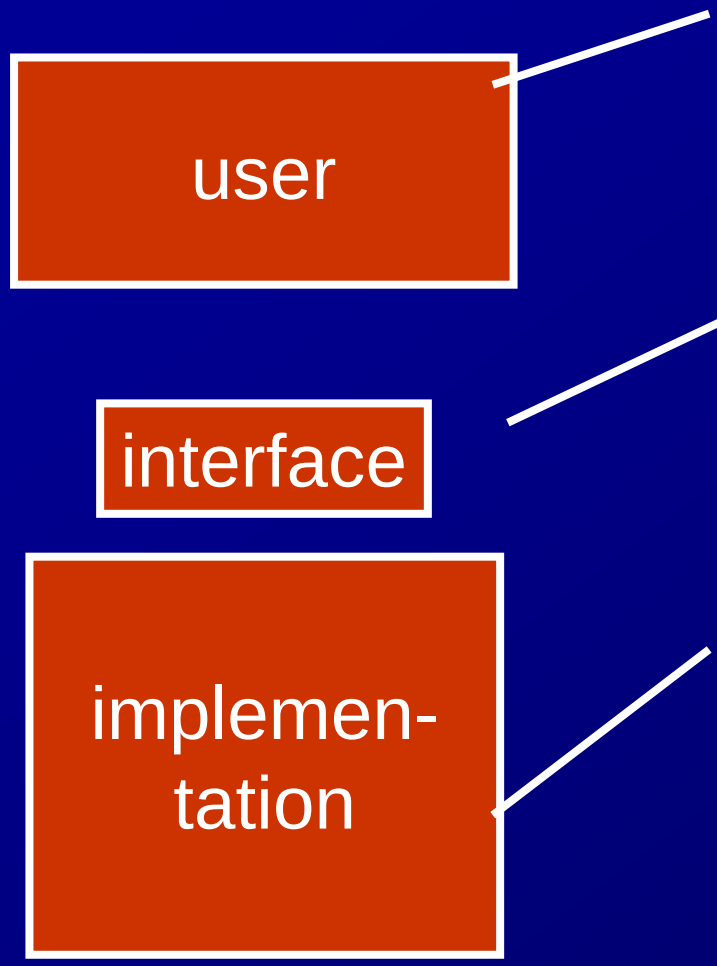
- Discuss what a module interface defines.
  - Note 1: a module has an interface and an implementation
  - Note 2: defines = what kind of information we should write
  - ➢ You can use the stack data structure as an example.

Internal data representation (Quiz #1)

- Internal representations are likely to change
- eg: stack
  - array + index
  - linked list

```
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;
}
```

Module mechanism in general



uses implementations through ***names*** in the IF

defines ***names*** of functions, variables and types

defines ***bodies*** of functions, variables and types
(covering those defined in the IF)

Module mechanism in C

Interface: *h (header file)
prototypes with extern

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

stack.c

implementation:
specify IF via #include
defines functions, vars.

```
#include "stack.h"
int corresponding(char* str) {
  stk s;
  stk_init(&s);
  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;
```

user:
specify via #include
use functions, vars.

Enforcement of the module mechanisms

- enforcing (i.e., information hiding): users can use only names in the interface
  - using other names causes an error
  - changes of non-exposed functions etc. won't affect the users
- not enforcing: interface serves merely as a guideline to the users

C: non-enforcing

- definitions in a module can be accessed from outside by writing extern decls.

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

Problem of modular data representation (Quiz #2)

users depend on internal representation

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

```
typedef struct stack { char* c; int t; } stk;
extern void stk_init(stk* s);
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;
void stk_init(stk* s) { ... }
void stk_push(stk* s, char c)
{ s->c[stk->t++] = c; }
char stk_pop(stk* s) { return s->c[--stk->t]; }
int stk_empty(stk* s) { return (s->t == 0); }
```

stack.c

Internal data representation should be hidden!

vs.

users have to use be able to access some data represent'n

Abstract data types[LZ74]

- Module mechanism for data types

- uses provided type names
- uses provided functions

- names of providing types and functions

- internal representation of types
- function bodies wrt representation

CLU

[LZ74, LAB+79]

data type definition

data type =
provided funcs.
+ internal repr.
+ impl. of funcs.

```
stack: cluster is push, pop, empty;
  rep =
    (tp: integer;
     stk: array[1..] of char;)
  create ... 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;
    ...
end stack
```

provided functions

internal repr.

impl. of functions

CLU

[LZ74, LAB+79]

data type definition

```
stack: cluster is push, pop, empty;
  rep =
    (tp: integer;
     stk: array[1..] of char;)
  create ... 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;
    ...
end stack
```

provided functions

internal repr.

impl. of functions

user

use of type

```
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;
```

Guarantees given by a module mechanism

- As long as:
  - the interface is not changed,
  - a module provider defines functions as defined by the interface, and
  - a module user uses only functions defined in the interface,

  a module system guarantees that the provider can **safely** change module implementations?

Safety guaranteed by a module mechanism

- A program can be executed
  - vs. modules are failed to be connected
- A program "runs"
  - vs. a function to be called is not defined
  - vs. when a function is called, the program stops

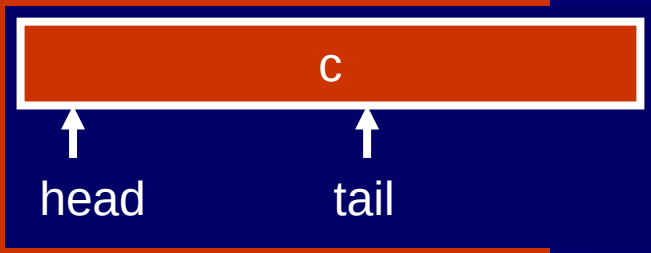
～roughly equivalent to be ***type safe***

type safety: when a program passes data,
the type of the data is the same one as expected

Is this a correct stack definition?

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 substitution principle [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.

Behavioral subtyping[LW94]

- An interface defines abstract states and behaviors of a module
  - Behaviors: precondition + postcondition
- Eg: a stack
  - (abstract state) l : list of elements
  - push(x) : (post cond.) $l_{post} = l_{pre} :: x$
  - pop(x) : (pre cond.) $l_{pre} \neq \varphi$
    (post cond.) ret = $last(l_{pre})$, $l_{post} = butlast(l_{pre})$

How behavioral subtyping is used

- Interface
  - Abstract state & behavior
- Module provider
  - defines correspondence between internal state and abstract state, and
  - defines functions so that "when each function is executed under a state where preconditions are satisfied, the post conditions will be satisfied"
- User

References

- [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.

情報隠蔽

- モジュールの実現を、モジュールの外側から隠す仕組み

Liskov置換原則 [LW94]

Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

問題

- () [ ] {} という三種類の括弧を含む文字列で、括弧が正しく対応しているかを判定するプログラムの概略を考えよ
  - 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
- スタック構造をモジュールとして定義する。インタフェースを定義せよ
- スタック構造の作り方には色々ある。あるスタック構造を別のスタック構造で置き換えても、スタック構造を使うモジュールが作しく動くことを保証するためには、どのようなことをすればよいか?

outline

- 例題
- Parnas原理に従った良いモジュールは、どのような作り方をされるべきか?
- 何らかのデータを表わすものの場合、どうやったら隠せるのか?
- 抽象データ型: データ = それに対する操作の集まり
- 正しい置き換えとは?
  - データ型による定義
  - 振舞部分型
- レコード型でやった場合と比べる

モジュール

ALGOL68 module

typical module systems

- module name
- exports
- use / imports
- interface
- implementation
- “signature”

例えばCやJAVAでは

- C:
  - interfaceはヘッダファイル
  - exportsはextern宣言
  - useは#include “ヘッダファイル”
- Java
  - interfaceはなし (クラス定義が代用)
  - exportsはpublic宣言
  - useはなし (importは単なる略記)

モジュールの交換可能性と部分型

- 型: 値の集合
- 部分型: 値の部分集合
- 整数 <= 実数
- 実数をしまう変数 = モジュールの関数
  - 実数を返す
  - 整数を返す

分割コンパイル

モジュールの交換可能性

- 置き換えてよい -> 置き換えても期待通りに動く
- 期待通りとは
  - コンパイルできる、はその一例
  - もっと正確に表わすことは?

抽象データ型

CLU言語

問題:「インタフェース」のあり方

- どう「規格化」するのか?

モジュール

- 定義 (一般的な「モジュール」):
  - 大きなシステムの部分を構成するもの
  - 他のモジュールとの連携方法が規格化されている

振舞部分型(behavioral subtyping)

```
stack = type
for all s : stack
  push = proc(i: int)
            ensures s_post.items =
  pop = proc() returns (int)
            requires s_pre.times !=
            ensures result = last(s_
                        s_post.items =
```

```
n
ir
ir
v
}
ir
}
```

```
module Stack

struct pair { int val; struct pair *next; } bot;
struct pair *top=&bot;
void push(int i) {
  struct pair *p = malloc(sizeof(struct pair));
  p->val = i;
  p->next = top;
  top = p;
}
int pop() {
  if (top != &bot) {
    int v = top->val;
    top = top->next;
  } else
    error(“cannot pop from an empty stack.”);
```

C言語のモジュール機構の問題

- データの型をインタフェースに定義しなければいけない
  - stack.h — struct stack を含む
  - 使用者: stackが文字配列 + 添字であることに依存してしまう
  - 提供者が内部表現(structの定義)を変更→使用者に影響
- スタック要素の型を変更→同様