Homework 7

ACM1501 U201514716

1. Consider what would happen if we pop an element from the empty stack when contracts are not checked in the linked list implementation? When does an error arise?

   top->next

2. Stacks are usually implemented with just one pointer in the header, to the top of the stack. Rewrite the implementation in this style, dispensing with the bottom pointer, terminating the list with NULL instead.

```
/*
 * Stacks of string
 *
 * 15-122 Principles of Imperative Computation */

typedef string elem;

/*** Interface ***/

// typedef _______ stack;
typedef struct stack_header* stack;

bool stack_empty(stack S); /* O(1) */
stack stack_new(); /* O(1) */
void push(stack S, elem x); /* O(1) */
elem pop(stack S) /* O(1) */
/*@requires !stack_empty(S); @*/ ;


/*** Implementation ***/

/* Aux structure of linked lists of integers */

struct list_node {
  elem data;
  struct list_node* next;
};
typedef struct list_node list;

bool is_cyclic(list* start, list* end) {
  if (start == NULL) return false;
  list* a = start->next;
  list* t = start;
```

```
  while (a != t) {
    if (a == NULL || a->next == NULL) return false;
    a = a->next->next;
    //@assert t != NULL;   // because achilles is faster and will hit NULL end quic
    t = t->next;
  }
  //@assert a == t;
  return true;
}

bool is_segment(list* start, list* end)
{
  if (is_cyclic(start, end)) return false;
  list* p = start;
  while (p != NULL) {
    if (p == end) return true;
    p = p->next;
    if (p == start) return false;
  }
  return false;
}


/* Stacks of elems */

struct stack_header {
  list* top;
};

bool is_stack(stack S) {
  if (S == NULL) return false;
  if (S->top == NULL || S->bottom == NULL) return false;
  if (!is_segment(S->top, S->bottom)) return false;
  return true;
}

bool stack_empty(stack S)
//@requires is_stack(S);
{
  return S->top->next == NULL;
}

stack stack_new()
//@ensures is_stack(\result);
//@ensures stack_empty(\result);
{
```

```
  stack S = alloc(struct stack_header);
  list* l = alloc(struct list_node); /* does not need to be initialized! */
  S->top = l;
  return S;
}

void push(stack S, elem x)
//@requires is_stack(S);
//@ensures is_stack(S);
{
  list* l = alloc(struct list_node);
  l->data = x;
  l->next = S->top;
  S->top = l;
}

elem pop(stack S)
//@requires is_stack(S);
//@requires !stack_empty(S);
//@ensures is_stack(S);
{
  elem e = S->top->data;
  S->top = S->top->next;
  return e;
}

stack stack_copy(stack S)
//@requires is_stack(S);
//@ensures is_stack(\result);
{
  stack S2 = alloc(struct stack_header);
  S2->top = S->top;
  return S2;
}
```

1. prog4

   clac.c0

```
/*
 * Clac, stack-based calculator language
 *
 * 15-122 Principles of Imperative Computation
 */

#use <string>
#use <parse>
```

```
#use <conio>
#use <args>
#use <util>

/* Return true: clac top-level interpreter will expect more input */
/* Return false: clac top-level interpreter will exit */
bool eval(queue Q, stack S)
//@ensures \result == false || queue_empty(Q);
{
  while (!queue_empty(Q)) {
    string tok = deq(Q); /* is this dequeue safe? */
    // print("Read: "); print(tok); print("\n");
    if (string_equal(tok, "print")) {

      /* next line is unsafe and should be fixed */
      int x = pop(S);
      printint(x); print("\n");
    } else if (string_equal(tok, "quit")) {
      return false;             /* do not continue */
    } else if (string_equal(tok, "+")) {
      int A = pop(S);
      int B = pop(S);
      push(S,A+B);
    } else if (string_equal(tok, "-")) {
      int A = pop(S);
      int B = pop(S);
      push(S,A-B);
    } else if (string_equal(tok, "*")) {
      int A = pop(S);
      int B = pop(S);
      push(S,A*B);
    } else if (string_equal(tok, "/")) {
      int A = pop(S);
      int B = pop(S);
      push(S,A/B);
    } else if (string_equal(tok, "%")) {
      int A = pop(S);
      int B = pop(S);
      push(S,A+B);
    } else if (string_equal(tok, "<")) {
      int A = pop(S);
      int B = pop(S);
      push(S,A<B?1:0);
    } else if (string_equal(tok, "drop")) {
      pop(S);
    } else if (string_equal(tok, "dup")) {
```

```
  int A = pop(S);
  push(S,A);
  push(S,A);
} else if (string_equal(tok, "swap")) {
  int A = pop(S);
  int B = pop(S);
  push(S,A);
  push(S,B);
} else if (string_equal(tok,"rot")) {
  int A = pop(S);
  int B = pop(S);
  int C = pop(S);
  push(S,B);
  push(S,A);
  push(S,C);
} else if (string_equal(tok, "skip")) {
  int n = pop(S);
  if (n < 0) error("Invalid negaive number: n");
  while (queue_empty(Q) && n != 0) {
    deq(Q);
    n--;
  }
  if (n != 0) error("Invalid size for 'skip'");
} else if (string_equal(tok, "if")) {
  int cnd = pop(S);
  int frec = 0;
  cnd == 0 ? 2 : 0;
  while (!queue_empty(Q) && frec != 0) {
    deq(Q);
    frec--;
  }
  if (frec != 0) error("Invalid size for 'if");
} else if (string_equal(tok, "else")) {
  if(!queue_empty(Q)){
    deq(Q);
  }else{
    error("Invalid 'else");
  }
} else if (string_equal(tok, "pick")) {
  stack T = stack_new();
  int n = pop(S);
  int temp = 0;
  int flag = 0;
  if(n > 0) flag = 1;
  while(!stack_empty(S) && n>0){
    temp = pop(S);
```

```
      push(T,temp);
      n--;
    }
    if(n != 0){
      error("Invalid 'pick'");
      flag = 0;
    }
    while(!stack_empty(T)){
      n = pop(T);
      push(S,n);
    }
    if(flag == 1) push(S,temp);
  } else {
    /* not defined as an operation name, should be an int */
    int* p = parse_int(tok, 10);
    if (p == NULL) { /* not an int */
      /* call error(msg) which prints msg and aborts */
      error(string_join("undefined token ", tok));
    }
    push(S, *p);
  }
 }
 return true;                  /* continue */
}
```