Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.087: Practical Programming in C

IAP 2010

Problem Set 6 – Solutions

Part 1: Pointers to pointers. Multidimensional arrays. Stacks and queues.

Out: Wednesday, January 20, 2010. **Due:** Friday, January 22, 2010.

This is Part 1 of a two-part assignment. Part 2 will be released Thursday.

Problem 6.1

In this problem, we will implement a simple "four-function" calculator using stacks and queues. This calculator takes as input a space-delimited infix expression (e.g. `3 + 4 * 7`), which you will convert to postfix notation and evaluate. There are four (binary) operators your calculator must handle: addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`). In addition, your calculator must handle the unary negation operator (also `-`). The usual order of operations is in effect:

- the unary negation operator `-` has higher precedence than the binary operators, and is evaluated right-to-left (*right-associative*)
- `*` and `/` have higher precedence than `+` and `-`
- all binary operators are evaluated left-to-right (*left-associative*)

To start, we will not consider parentheses in our expressions. The code is started for you in `prob1.c`, which is available for download from Stellar. Read over the file, paying special attention to the data structures used for tokens, the stack, and queue, as well as the functions you will complete.

(a) We have provided code to translate the string to a queue of tokens, arranged in infix (natural) order. You must:

- fill in the `infix_to_postfix()` function to construct a queue of tokens arranged in postfix order (the infix queue should be empty when you're done)
- complete the `evaluate_postfix()` function to evaluate the expression stored in the postfix queue and return the answer

You may assume the input is a valid infix expression, but it is good practice to make your code robust by handling possible errors (e.g. not enough tokens) . Turn in a printout of your code, along with a printout showing the output from your program for a few test cases (infix expressions of your choosing) to demonstrate it works properly.

Answer: Here's one possible implementation: (only functions `infix_to_postfix()` and `evaluate_postfix()` shown)

```
/* creates a queue of tokens in postfix order from a queue of tokens in infix order */
/* postcondition: returned queue contains all the tokens, and pqueue_infix should be
   empty */
struct token_queue infix_to_postfix(struct token_queue * pqueue_infix) {
  /* TODO: construct postfix-ordered queue from infix-ordered queue;
     all tokens from infix queue should be added to postfix queue or freed */
  p_expr_token stack_top = NULL, ptoken;
  struct token_queue queue_postfix;

  queue_postfix.front = queue_postfix.back = NULL;
  for (ptoken = dequeue(pqueue_infix); ptoken; ptoken = dequeue(pqueue_infix)) {
    switch (ptoken->type) {
    case OPERAND:
      /* operands added directly to postfix queue */
      enqueue(&queue_postfix, ptoken);
      break;
    case OPERATOR:
      /* operator added to stack, after operators of higher
         precedence are moved to queue */
      while (stack_top &&
        (op_precedences[stack_top->value.op_code] >
        op_precedences[ptoken->value.op_code] ||
        (op_precedences[stack_top->value.op_code] ==
        op_precedences[ptoken->value.op_code] &&
        op_associativity[op_precedences[ptoken->value.op_code]] == LEFT)))
        enqueue(&queue_postfix, pop(&stack_top));
      push(&stack_top, ptoken);
      break;
    default: /* other tokens ignored */
      free(ptoken);
      break;
    }
  }
  while (stack_top) /* pop remaining operators off stack */
    enqueue(&queue_postfix, pop(&stack_top));
  return queue_postfix;
}

/* evalutes the postfix expression stored in the queue */
/* postcondition: returned value is final answer, and pqueue_postfix should be empty */
double evaluate_postfix(struct token_queue * pqueue_postfix) {
  /* TODO: process postfix-ordered queue and return final answer;
     all tokens from postfix-ordered queue is freed */
  double ans = 0.;
  p_expr_token stack_values = NULL, ptoken, pvalue;
  double operands[2]; /* max two operands */
  union token_value value;
  int i;

  while ( (ptoken = dequeue(pqueue_postfix)) ) {
    switch (ptoken->type) {
    case OPERAND:
      /* operands always pushed to stack */
      push(&stack_values, ptoken);
      break;
    case OPERATOR:
```

```
        /* pop operands from stack */
        for (i = 0; i < op_operands[ptoken->value.op_code]; i++) {
          if ( (pvalue = pop(&stack_values)) ) {
            operands[i] = pvalue->value.operand;
            free(pvalue); /* done with token */
          } else
            goto error;
        }
        /* process operands according to opcode */
        /* note operands are popped in reverse order */
        switch (ptoken->value.op_code) {
        case ADD:
          value.operand = operands[1]+operands[0];
          break;
        case SUBTRACT:
          value.operand = operands[1]-operands[0];
          break;
        case MULTIPLY:
          value.operand = operands[1]*operands[0];
          break;
        case DIVIDE:
          value.operand = operands[1]/operands[0];
          break;
        case NEGATE:
          value.operand = -operands[0];
        }
        /* push new token with operator result to stack */
        push(&stack_values, new_token(OPERAND, value));
      default:
        free(ptoken); /* free token */
        break;
      }
    }
    /* return value is on top of stack (should be last value on stack) */
    if ( stack_values )
      ans = stack_values->value.operand;

  cleanup:
    /* free any remaining tokens */
    while ( (ptoken = dequeue(pqueue_postfix)) )
      free(ptoken);
    while ( (pvalue = pop(&stack_values)) )
      free(pvalue);
    return ans;

  error:
    fputs("Error evaluating the expression.\n", stderr);
    goto cleanup;
  }
```

(b) Now, an infix calculator really is not complete if parentheses are not allowed. So, in this part, update the function `infix_to_postfix()` to handle parentheses as we discussed in class. Note: your postfix queue should contain no parentheses tokens. Turn in a printout of your code, along with a printout showing the output from your program for a few test cases utilizing parentheses.

Answer: Here's an implementation: (only function `infix_to_postfix()` shown)

```
/* creates a queue of tokens in postfix order from a queue of tokens in infix order */
/* postcondition: returned queue contains all the tokens, and pqueue_infix should be
   empty */
struct token_queue infix_to_postfix(struct token_queue * pqueue_infix) {
  /* TODO: construct postfix-ordered queue from infix-ordered queue;
     all tokens from infix queue should be added to postfix queue or freed */
  p_expr_token stack_top = NULL, ptoken;
  struct token_queue queue_postfix;

  queue_postfix.front = queue_postfix.back = NULL;
  for (ptoken = dequeue(pqueue_infix); ptoken; ptoken = dequeue(pqueue_infix)) {
    switch (ptoken->type) {
    case OPERAND:
      /* operands added directly to postfix queue */
      enqueue(&queue_postfix, ptoken);
      break;
    case OPERATOR:
      /* operator added to stack, after operators of higher
         precedence are moved to queue */
      while (stack_top && stack_top->type == OPERATOR &&
        (op_precedences[stack_top->value.op_code] >
        op_precedences[ptoken->value.op_code] ||
        (op_precedences[stack_top->value.op_code] ==
        op_precedences[ptoken->value.op_code] &&
        op_associativity[op_precedences[ptoken->value.op_code]] == LEFT)))
        enqueue(&queue_postfix, pop(&stack_top));
      push(&stack_top, ptoken);
      break;
    case LPARENS:
      /* pushed to operator stack */
      push(&stack_top, ptoken);
      break;
    case RPARENS:
      /* pop operators off stack until left parentheses reached */
      free(ptoken); /* parentheses not included in postfix queue */
      while ( (ptoken = pop(&stack_top)) ) {
        if (ptoken->type == LPARENS) {
          free(ptoken);
          break;
        }
        enqueue(&queue_postfix, ptoken);
      }
    }
  }
  while (stack_top) /* pop remaining operators off stack */
    enqueue(&queue_postfix, pop(&stack_top));
  return queue_postfix;
}
```

Problem 6.2

A useful data structure for storing lots of strings is the "trie." This tree structure has the special property that a node's key is a prefix of the keys of its children. For instance, if we associate a node with the string "a," that node may have a child node with the key "an," which in turn may have a child node "any." When many strings share a common prefix, this structure is a very inexpensive

way to store them. Another consequence of this storage method is that the trie supports very fast searching – the complexity of finding a string with m characters is O(m).

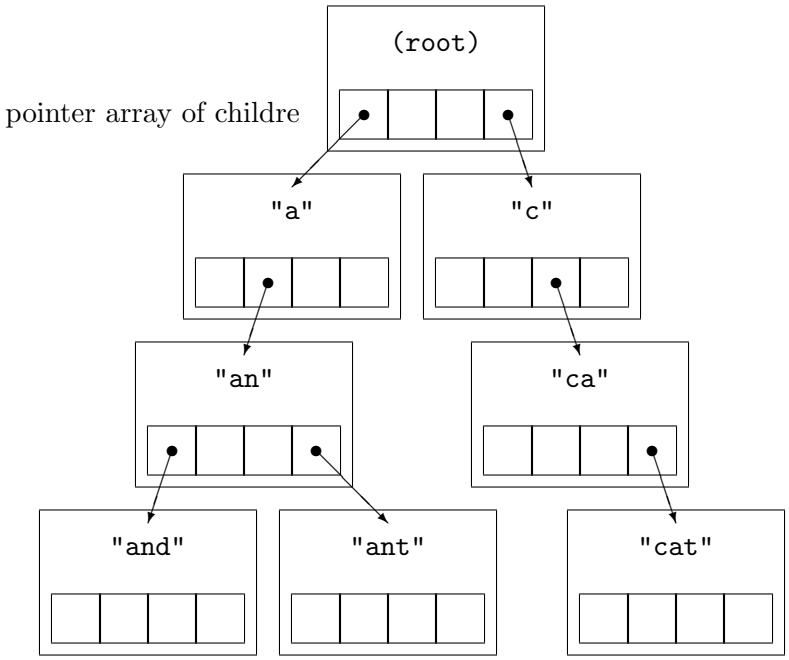

Figure 6.2-1: Trie structure (translations not shown). For each node, its key (e.g. "and") is not explicitly stored in the node; instead, the key is defined by the node's position in the tree: the key of its parent node + its index in that parent's pointer array of children.

In this problem, you will utilize a trie structure and to implement a simple one-way English-to-French dictionary. The trie structure, shown in Figure 6.2-1, consists of nodes, each of which contains a string for storing translations for the word specified at that node and an array of pointers to child nodes. Each node, by virtue of its position in the trie, is associated with a string; in the dictionary context, this string is the word (or part of a word) in the dictionary. The dictionary may contain multiple translations for the same word; in this case, words should be separated by commas. For example: the word like, which has two meanings, could translate as *comme*, a preposition, or as *aimer*, a verb. Thus, the translation string should be "comme,aimer." To get you started, we've provided code in `prob2.c`, which can be downloaded from Stellar. You will need to:

- fill in the helper functions `new_node()`, `delete_node()`
- complete the function `add_word()`, which adds a word to the trie
- complete the function `lookup_word()`, which searches the trie for a word and returns its translation(s)

Once your code is working, run a few test cases. Hand in a copy of your code, and a printout of your program running in gdb, with a few example translations to demonstrate functionality.

Answer: one possible implementation of the four functions is shown below:

```
/* allocate new node on the heap
   output: pointer to new node (must be freed) */
struct s_trie_node * new_node(void) {
  /* TODO: allocate a new node on the heap, and
     initialize all fields to default values */
  struct s_trie_node * pnode =
      (struct s_trie_node *)malloc(
          sizeof(struct s_trie_node));
  int i;

  pnode->translation = NULL;
  for (i = 0; i < UCHAR_MAX+1; i++)
    pnode->children[i] = NULL;
  return pnode;
}

/* delete node and all its children
   input: pointer to node to delete
   postcondition: node and children are freed */
void delete_node(struct s_trie_node * pnode) {
  /* TODO: delete node and all its children
     Be sure to free non-null translations!
     Hint: use recursion
   */
  int i;

  if (pnode->translation)
    free(pnode->translation);
  for (i = 0; i < UCHAR_MAX+1; i++)
    if (pnode->children[i])
      delete_node(pnode->children[i]);
  free(pnode);
}

/* add word to trie, with translation
   input: word and translation
   output: non-zero if new node added, zero otherwise
   postcondition: word exists in trie */
int add_word(const char * word, char * translation) {
  /* TODO: add word to trie structure
     If word exists, append translation to existing
     string
     Be sure to store a copy of translation, since
     the string is reused by load_dictionary()
   */
  struct s_trie_node * pnode = proot;
  int i, len = strlen(word), inew = 0;
  unsigned char j;

  for (i = 0; i < len; i++) {
    j = word[i];
    if ((inew = !pnode->children[j]))
      pnode->children[j] = new_node();
    pnode = pnode->children[j];
  }
  if (pnode->translation) {
    /* concatenate strings */
```

```
    char * oldtranslation = pnode->translation;
    int oldlen = strlen(oldtranslation),
        newlen = strlen(translation);
    pnode->translation = malloc(oldlen + newlen + 2);
    strcpy(pnode->translation, oldtranslation);
    strcpy(pnode->translation+oldlen,",");
    strcpy(pnode->translation+oldlen+1,translation);
    free(oldtranslation);
  } else
    pnode->translation = strcpy(malloc(
        strlen(translation)+1),translation);
  return inew;
}

/* search trie structure for word and return translations
   input: word to search
   output: translation, or NULL if not found */
char * lookup_word(const char * word) {
  /* TODO: search trie structure for word
     return NULL if word is not found */
  struct s_trie_node * pnode = proot;
  int i, len = strlen(word);
  unsigned char j;

  for (i = 0; i < len; i++) {
    j = word[i];
    if (!pnode->children[j])
      return NULL;
    pnode = pnode->children[j];
  }
  return pnode->translation;
}
```

6.087 Practical Programming in C

January (IAP) 2010