2003/2004 SOUTHERN CALIFORNIA REGIONAL
ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST

Problem 3
Refactor

PIT Corporation is creating a comprehensive Interactive Development Environment that will revolutionize programming. A key feature of the IDE is the semantics-aware source code editor. You have been contracted to prototype the variable renaming editing operation rapidly. While the finished editor will support many languages, the prototype editor only supports C–, a small subset of C, which only features simple variable declarations, expressions, and statements. It does not contain comments or preprocessor directives.

Input will be a C– source program consisting of the text "**`main()`**" on the first line by itself, followed by one *compound-statement*, followed by "**`#if 0`**" on a line by itself, one or more semantic editing operations on separate lines, followed by "**`#endif`**" on a line by itself. Editing operations are applied in the order they appear. The format of the "rename variable" operation is

*n***`R/`***old_variable_name***`/`***new_variable_name***`/`**

where *n* is a line number in the source code containing the variable to be renamed, with slashes delimiting old and new names. Renaming the variable will change it where it is declared and everywhere it is used. This is not a global string replace. It does not change identically named variables declared in a different scope.

Note: The 'main()', '#if 0', and '#endif' lines above are not part of the C– language grammar. They just allow the input for this problem to be treated as a valid C program.

Notes: *old_variable_name* refers to the first occurrence on the specified line number, *n*, if it occurs more than once. To avoid conflicts, *new_variable_name* will not be the same name as a previously declared variable.

Output will be the updated source code after application of the semantic editing commands without the lines "**`#if 0`**" through "**`#endif`**". Preserve whitespace from the input in the output.

Sample Input

```
main()
{
  int  tmp ;
  int a;  short b;
  a=1;  b=4;  tmp=10;
  {
    int tmp;
    tmp=b;
    b=a;
    a=tmp;
  }
  tmp = tmp + b;
  printf("a=%d b=%d tmp=%d\n",a,b,tmp);
}
#if 0
10R/tmp/hold/
7R/hold/scratch/
12R/tmp/sum/
#endif
```

Problem 3
Refactor (continued)

Output for the Sample Input

```
main()
{
  int  sum ;
  int a;  short b;
  a=1;  b=4;  sum=10;
  {
    int scratch;
    scratch=b;
    b=a;
    a=scratch;
  }
  sum = sum + b;
  printf("a=%d b=%d tmp=%d\n",a,b,sum);
}
```

C– GRAMMAR

The grammar is in the form used in "The C Programming Language" by Kernighan and Ritchie.

Syntactic categories (non-terminals) are indicated by *italic* type, and literal words and characters (terminals) in **`typewriter`** style. Alternative categories are listed on separate lines. An optional terminal or optional nonterminal symbol carries the subscript "$_{opt}$".

declaration:
    type-specifier declarator **`;`**
declaration-list:
    declaration
    declaration-list declaration
type-specifier:
    `short`
    `int`
declarator:
    direct-declarator
direct-declarator:
    identifier
compound-statement:
    `{` *declaration-list*$_{opt}$ *statement-list*$_{opt}$ **`}`**
statement:
    expression-statement
    compound-statement
expression-statement:
    expression$_{opt}$ **`;`**
statement-list:
    statement
    statement-list statement
expression:
    assignment-expression
assignment-expression:
    additive-expression
    unary-expression assignment-operator assignment-expression
assignment-operator:
    `=`

Problem 3
Refactor (still continued)

additive-expression:
    multiplicative-expression
    additive-expression **+** *multiplicative-expression*
    additive-expression **-** *multiplicative-expression*
multiplicative-expression:
    unary-expression
    multiplicative-expression ***** *unary-expression*
    multiplicative-expression **/** *unary-expression*
unary-expression:
    postfix-expression
postfix-expression:
    primary-expression
    postfix-expression **(** *argument-expression-list*$_{opt}$ **)**
primary-expression:
    identifier
    constant
    string
    (*expression* **)**
argument-expression-list:
    assignment-expression
    argument-expression-list **,** *assignment-expression*
constant:
    integer-constant
identifier: An alphabetic character followed by zero-to-30 alphanumeric characters.
    [a-zA-Z][a-zA-Z0-9]{0,30}
integer-constant: One-to-nine decimal digits.
    [0-9]{1,9}
string: Double quote, zero to 200 printable characters (excluding **"**), terminating double quote.
    "[!#$%&'()*+,-./0-9:;<=>?@A-Z[\]^_‘a-z{|}~]{0,200}**"**

The keywords **short** and **int** are reserved words, and will not be used other than as *type-specifiers*.