

**The University of Texas at San Antonio  
Computer Science Program  
San Antonio, Texas 78249**

**CS 1723, Data Structures  
Spring Semester, 1997  
Programming Assignment 3, Due February 19, 1997  
*Translation to Reverse Polish***

**The Assignment:** For this assignment you will write *two* separate programs. Together, these should evaluate an input arithmetic expression. The first program, `translate`, will translate the arithmetic expression to *reverse Polish* form. The second program, `evaluate`, will find the value of the reverse Polish produced by the first program. For the final runs, you can pipe the output of the first in as input to the second, so that you use the command:

```
runner% translate <sourcefile | evaluate
```

The input source will be made up of the following elements:

- single-digit integer constants, 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9.
- single-letter variables, like `a` or `X`. (Not used by our evaluator. See “Extra Credit” at the end.)
- operators: `^`, `*`, `/`, `+`, or `-` (or `=` for the Extra Credit part).
- parentheses: `(` and `)`.
- the special symbols: `#` to terminate the whole input (and `;` to terminate an expression in the Extra Credit part).

This assignment limits to single-digit integer constants and single-character variables in order to keep things simpler.

For example, here is sample input source, which calculates one root of the equation  $y = x^2 + 3x + 2$ , with  $a = 1$ ,  $b = 3$ ,  $c = 2$ :

```
((b^2 - 4*a*c)^(1/2) - b)/(2*a)#
```

or with specific constants:

```
((3^2 - 4*1*2)^(1/2) - 3)/(2*1)#
```

Notice that all *whitespace* (= space, newline, tab, etc.) should be ignored on input.

**Overall organization:** Both programs should be contained in a single directory, say `assign3`. Within this directory there will be several files implementing the two programs, but there will only be *one* `makefile`, described below. Note that there will be *two* `main()` functions, corresponding to the two programs.

**The first program `translate.c`:** Organize the program as (at least) *three* files: `translate.c`, `tstack.h`, and `tstack.c`.

The `tstack.h` and `tstack.c` files should implement a stack of characters, and for this you *must* use a stack based on *pointers*, *linked lists*, and *dynamic storage allocation*, as described in the blue book and in class. (You will probably want to add a function `top()` that will return the topmost character without popping it.)

In addition to `main()`, the `translate.c` file should contain functions with prototypes

```
int checkprec (char c, int *prec, char *assoc);
int getnext(void);
```

- The function `checkprec()` should return the precedence and associativity of the operator stored in the variable “c”, according to the following table:

Operator	Precedence	Associativity
$\wedge$	5	R
*	4	L
/	4	L
+	3	L
-	3	L
=	2	R
(	1	R
)	1	R

There *must* be a static array of structures inside `checkprec()` which stores this information. This structure *must* be initialized using a list of initializers enclosed in braces, exactly as shown in Section 6.3 of the white book (page 133). (The = operator is only used in the Extra Credit part of the assignment.)

- The function `getnext()` should return the next non-whitespace character. (Use `isspace()` in `<ctype.h>`.)

The translation algorithm will be discussed in class. Notice that for each non-whitespace character, the program must decide whether to output it or to stack it. Operands (digits and letters) are always output immediately. Operators (including parentheses) are always stacked, but in some cases not before one or more other operators are popped and output. The following rules apply:

1. Always push a left parenthesis.
2. Always push an operator if the stack is empty, if an operator of lower precedence is on top, or if an operator of equal precedence is on top and right-to-left associativity applies.
3. Otherwise pop and output the stacktop, and try rules 1 and 2 again.
4. A left and a right parenthesis (right above left) on the stack are always simply deleted.
5. A # character terminates the current input. Output a # and terminate the program.

The result of the sample input above should be:

```
b2^4a*c*-12/^b-2a*/#
32^41*2*-12/^3-21*/#
```

**The initial makefile for translate.c:**

```
# makefile for the translate program
translate: translate.c tstack.c tstack.h
        cc -g -o translate translate.c tstack.c
tlint:
        lint -m -u translate.c tstack.c
```

**The second program evaluate.c:** Organize this second program as (at least) *three* files: `evaluate.c`, `estack.h`, and `estack.c`.

The evaluation of one of these RPN strings can proceed as described in the text and in class: Use an evaluation stack. Operands are pushed on the stack, and operators pop their arguments off the stack and push the result. When the final # is encountered, the remainder of the stack is popped and printed. (It should just be a single value.)

The arithmetic should be done using doubles. The stack *must* be implemented using linked lists like the other stack. (Notice that this stack holds doubles, while the other stack holds characters.) Thus the output of the second example should be the following: (Unless you are doing the Extra Credit part, the first example uses variable names which will not have a value and so cannot be evaluated.)

```
-1.000000
```

A single character that is a digit can be converted to a double with

```
#include <ctype.h>
...
if (isdigit(c))
    return (double) (c - '0');
...
```

The raise-to-a-power operator can be handled with the built-in function `pow(x, y)` (see the white book, page 251). For this to work you need to include `<math.h>`, and (on ringer) you need to add `-lm` to the compiler options, so that the math library will be searched, as shown in the makefile below. The `lint` program also needs this option as shown—without it `lint` will produce 600 lines of warnings.

**The new makefile for both `translate.c` and `evaluate.c`:**

```
# makefile for translate and evaluate programs
all:  translate evaluate
translate: translate.c tstack.c tstack.h
        cc -g -o translate translate.c tstack.c
evaluate:  evaluate.c estack.c estack.h
        cc -g -o evaluate evaluate.c estack.c -lm
tlint:
        lint -m -u translate.c tstack.c
elint:
        lint -m -u evaluate.c estack.c -lm
```

**Required Execution:** It will be enough to successfully execute the test program above, in two ways, once showing the intermediate RPN, and once using a pipe between `translate` and `evaluate`. However, for initial runs, you should test these separately and with very simple inputs. (Don't try the full input on the first run, and debug the two parts separately.)

**Extra Credit, if Bored with the Assignment:** You should include the assignment operator, so that your input can be a *sequence* of expressions, each separated by a semicolon, but with the whole still terminated by a `#`. Translation is similar, except that you flush the stack and start over with each `;`. For evaluation, one assumes that each variable is given a value before its use. During evaluation you will need to maintain a table of values of variables. An assignment causes a variable to take on a value. An expression without an assignment operator has its value printed. The hardest part of this extra credit part is maintaining the table of values of variables and keeping track of things on the evaluation stack, which you may want to expand to hold more information. Here is more elaborate sample input:

```
a = 1; b = 3; c = 2;
d = (b^2 - 4*a*c)^(1/2);
r = (0 - b + d)/(2*a);
s = (0 - b - d)/(2*a);
r; s;#
```

For this, the final evaluator would produce `-2.0000` and `-1.0000`, while the intermediate reverse Polish form would be:

```
a1=b03--=c2=db2^4a*c*-12/^=r0b-d+2a*/=s0b-d-2a*/=rs#
```