Assignment #3

Due Date: Thursday, 6 August, 1992

28 July 1992

You will write a program which reads arithmetic expressions, evaluates them, and prints the result.

1 Requirements

Your program will implement a reverse Polish notation calculator. The program will read from the standard input. Your program will compute the values of all expressions entered. Expressions will consist of integers and operators. Your program will recognize the operators `+`, `-`, `*`, `/`, and `%`. Furthermore, your program will print out the most recently computed result when it sees the operator `=`. Your program will discard the most recently computed result when it sees the operator `.`.

Your program will not place any arbitrary limits on the complexity of the expressions the user can evaluate. Your program will handle error conditions as gracefully as possible and will print appropriate error messages if memory is exhausted or if the expression the user enters contains an error.

1.1 Example

Indented lines represent user input; other lines represent output.

```
        2 3 *
        =
6
        4 + =
10
        7 - =
3
        5 6 * = + =
30
33
        12 3 * =
36
        .
        =
33
        12 12 12 * * 1 + =
1729
        . =
33
        .
        =
(stack empty)
```

2 What to Hand in

You should hand in a disk with your source code and an executable, and any special instructions to me. You may choose to supply a log file that demonstrates your program, if you like. I will not accept paper copies of solutions to this assignment.

3 Assignment-Specific Points

I will award seven points for a working RPN calculator, and seven points for arranging that the calculator can handle arbitrarily complicated expressions.

4 About Reverse Polish Notation

Evaluating ordinary arithmetic expressions is a pain, because you don't get to evaluate things in the order they're entered. To evaluate the expression `4 * ( 3 + 7 - 6 * 2) + 11` you have to skip forward to the part in the parentheses, then skip back; you can't evaluate it from left to right.

Fortunately there is an easier way, adopted by programming language designers, calculator manufacturers, logicians, and us. We will require that our expressions be entered in *reverse Polish notation*. What this means is that instead of each operator coming between the two quantities it's supposed to operate on (which is ambiguous), each operator follows the two quantities it's supposed to operate on.

A reverse Polish notation expression, therefore, is either just a number, or two simpler expressions *followed* by an operator symbol, which says how to combine the values of the two expressions. For example:

- The value of `2 3 +` is 5.
- The value of `3 4 *` is 12.
- The value of `2 3 + 3 4 * +` is 17; the final `+` says to evaluate the two expressions (`2 3 +` and `3 4 *`) that came before and add their values together.
- The value of `2 3 + 4 *` is 20; the final `*` says to take the values of the two preceding expressions, `2 3 +` and `4`, and multiply them.
- The value of `2 3 4 * +` is 14; the final `+` says to take the values of the two preceding expressions, `2` and `3 4 *`, and add them.

Issues of precedence and parenthesization disappear, because it's no longer ambiguous where a subexpression ends, and therefore it's clear what the operands of each operator symbol are. Every expression and sub-expression ends with an operator symbol.

Reverse polish notation is great for calculators because you get to write the operators in the order that the operations are actually performed. If you heed to compute two quantities, x and y, and add them together, then you first compute x, then compute y, and then do the addition—of course it's absurd to say you want to do the addition before you've computed y; that's impossible. Reverse Polish notation lets you actually write the expressions in the order they're computed: $x\ y\ +$ means 'compute x; then compute y; then add those two values.'

Since the operations in an RPN expression are written in the order that they actually have to be performed, it turns out that interpreting and evaluating these expressions is very easy. There's a natural way to implement an RPN calculator, which is the subject of the next section.

5 Stacks

The name *stack* is supposed to make you think of those stacks of plates with springs underneath that you sometimes find in cafeterias. The spring is under a whole stack of plates, but it keeps them at the right height so that only the top plate is visible. You can only see the top plate; you can only remove the top plate; but when you take the top plate off, the plate just below it appears and then you can examine or remove that plate. When you put a new plate on top, it obscures the plate that used to be visible.

So a stack is a data structure that has two operations defined on it: You can *pop* the stack; that means you take the top element off the stack and examine it; the element under the top element becomes the new top. We also speak of popping the top element itself. You can also *push* a new element onto the stack. If you push `foo` onto a stack and then push `bar`, `bar` is on the top of the stack and `foo` is under that; if you pop the stack you get `bar`, and then if you pop it again you get `foo`. The only way to examine the data at the bottom of the stack is to pop off everything else above it. If you try to pop the stack when the stack is empty, you get an error.

It turns out that a stack is just the thing for implementing an RPN calculator. The rules are simple:

1. Read the input from left to right.
2. If you see a number, push it onto the stack.
3. If you see an operator, pop the right number of operands off the stack, operate on them, and push the result on the stack.

For example, let's say the user enters the expression `2 3 * 4 +`. We should compute the value 10. What do we do? First, we see `2`, so we push that on the stack; then we push the `3` after it. Then we see the `*`, so we pop the `3` and the `2` from the stack, multiply them, and push the result, `6`. Then we push the `4`; the stack now contains a `6` and a `4` with the `4` at the top. Then we see the `+`; we pop the `6` and the `4`, add them, and push the result, 10, back on the stack. The top of the stack now contains the result `10`, which was the value we wanted to compute.

Let's do another example: `2 3 4 + *`. We see the `2`, the `3`, and the `4`, and push them on the stack in that order, so that the `4` is at the top and the `2` is at the bottom. Then we see the `+`; we pop the top two elements (The `4` and the `3` off the stack, add them, and push the result, 7, back on the stack. The stack now contains two elements, `2` and 7, with the 7 on top. Then we see the `*`, pop the two elements off the stack, multiply them, and push the result, 14, back on. The top [of the stack now contains the value 14, which is what we wanted to compute.

5.1 A Simple Implementation of a Stack

Here's code that implements a stack of <`int`>s and two functions for pushing and popping.

```
#define MAXDEPTH 100              /* Stack can contain up to 100 integers */

int stack[MAXDEPTH];
int stackpointer = 0;             /* Index of next empty stack element */

int pop(void)
{
  if (stackpointer == 0) {        /* Stack is empty */
    fprintf(stderr, "The stack is empty!\n");
    exit(1);
  }

  return stack[--stackpointer];
}

void push(int val)
{
  if (stackpointer > MAXDEPTH) {         /* Stack is full */
    fprintf(stderr, "Stack overflow!\n");
    exit(1);
  }

  stack[stackpointer++] = val;
}
```

Unfortunately, this is a little too simple. This program terminates if the user tries to make the stack too deep by computing too many intermediate results. Your program will not have this problem. We will arrange for our stack to grow as large as necessary, until the computer runs out of memory. Probably the simplest way to do this is to implement your stack as a linked list.

If we don't discuss linked lists in class by the end of the 29th of July, please mention it to me and I'll hand out detailed information about them on the 30th.