A very simple calculator

Daniel Bastos
dbastos@toledo.com

October 13th, 2006

Abstract

This is a small language for arithmetic, useful for those everyday calculations. I use it constantly from the GNU EMACS so that I can take advantage of a readline-like type of user interface which this program lacks. What I present here is some notes to myself about the behavior of the software and its implementation.

1 Introduction

1.1 History: changes from 0.4.

The current release date is "Fri Oct 05 21:49:16 EDT 2007". Statements can now break into more than one line.

1.2 An overview of the grammar

The software starts in `grammar.y`. We see a simple arithmetic grammar with the operators `=`, `+`, `-`, `*`, `/`, `%`, the unary `+`, the unary `-` and finally `^`. The predence is actually the inverse order we listed here.

The associativity of each is given by `grammar.y`. For example, `^` is right associative. The operator `=` is also right associative. All the others are left. So $1 - 2 - 3 = (1 - 2) - 3 = -4$ and not 2. Defining an associative direction for addition is not necessary, but we do it anyway so that we don't clash with `yacc`.

1.3 An overview of some of the features

We may separate statements with a semi colon, much like the C language. We can then write

```
x = 1; y = x^2 + 67; pi * y^2;
```

instead of separating each statement in a single line. If you want to convert radians to degrees, you can multiply it by `deg`. For example, `pi/2 * deg` equals 90.

Assignments don't get printed because we alredy know their values. Everything that gets printed also gets assigned to the variable `p`. We can then implement Newton's Method in the following way. Suppose we want to find the real root of

$$p(x) = x^3 + x + 1.$$

What's $p(-1)$? It's $-1 - 1 + 1 = -1$. So $x = -1$ gives a value of $p(x)$ close to zero, but underneath the x-axis. Since $p(x)$ is continuous, we may expect it to keep increasing and cross the x-axis a little further than -1. So we'll give Newton's method the value -1 and hope that it will converge to the root — it will.

The derivative of $p(x)$ is $p'(x) = 3x^2 + 1$. So we write

```
%./compute
-1;
  -1

- (p^3 + p + 1)/(3*p^2 + 1) + p;
  -0.75

- (p^3 + p + 1)/(3*p^2 + 1) + p;
  -0.68604651

- (p^3 + p + 1)/(3*p^2 + 1) + p;
  -0.68233958

- (p^3 + p + 1)/(3*p^2 + 1) + p;
  -0.6823278

- (p^3 + p + 1)/(3*p^2 + 1) + p;
  -0.6823278
```

So the root is some real number close to -0.6823278. I had to type -1 first because p is initially assigned the value 0.0.

1.4 Some behaviors and restrictions

Assignments do not affect p. You may not write to p either. It is read-only. Also e, deg, and π are read-only. All other variable names can be freely used, but I put a limit of 15 characters per variable. So, for example

```
%./compute
xxxxxxxxxxxxxxx = 1;
xxxxxxxxxxxxxxx;
  1
```

```
yyyyyyyyyyyyyyyy = 2;
./compute: token too long near line 3
./compute: syntax error near line 3
```

We could perhaps truncate and ignore the rest of the token, but I chose to be strict — it freed me from dealing with more memory allocation concerns.

1.5 Symbols

The stack managed by `yacc` has the type defined as a `union` of a `double` and a pointer to a `symbol`. It's a pointer to a `symbol` because, before we think of a `symbol` on the stack, we must load it in the `symbol` list. We say "symbol list" because we actually have a linked list of symbols, not a hash table.

A `symbol` is a `struct` composed of name, type, and a `union` of a `double` and a pointer to a function returning `double`. We also need a link to the next `symbol` in the list.

The installing of symbols is done by the `install` function also defined in `symbol.c`. We do not free any memory allocated by `install` because usually when we use a variable x, or y, we will keep using it until we finish the program. It may be interesting to reset values, but that's easily done by a simple assignment such as $x = 0$. The installing of new symbols is done by the lexical analyser, as input is read. If a variable is read, but not found in the `symbol` list, then it's a new `symbol` and the installing takes place.

1.6 The lexical analyser

Blanks, tabs and new lines are ignored. Numbers either start with a dot or with a digit. Once a number has been completely read, we would have read the next character of the input, so we just throw it back in the buffer.

If we read a `symbol`, then we will slowly read it one by one into its buffer whose maximum size is `SYMSIZE`. We add a byte for the 0-terminator. A `symbol` must start with a letter, so `isalpha` is used, but after the first, numbers may come, so the `while` loop uses `isalnum`. The last verification in the `while` loop is simply the size restriction. We do not want symbols bigger than `SYMSIZE`. The verification is done with pointer arithmetic since each letter takes a byte and so subtraction is convenient.

1.7 List of functions and operators

We currently support the following operators: `=`, `+`, `-`, `*`, `/`, `%`, `^`, `=\`, `!`, `<`, `<=`, `>`, `>=`, and the unaries `+` and `-`. The following constants have been defined: `pi`, `e`, `deg`, `p`. The following built in functions are defined: `sin`, `cos`, `atan`, `log`, `log10`, `exp`, `sqrt`, `int`, `abs`, `rand`, `fact`.