An Arithmetic Machine

Daniel C. Bastos
dbastos@math.utoledo.edu

October 13th, 2006

Abstract

This is an arithmetic machine; it performs arithmetic. Here you find some notes about its behavior and implementation.

1 Introduction

This software is useful for those everyday calculations; I use it constantly from my GNU emacs so that I can take advantage of a readline-like type of user interface which this program lacks, as it should. To achieve the same capability on a terminal, make sure to get yourself acquainted with a software such as `rlwrap`.

1.1 History: chances from 0.41

The current release date for version 0.42 is "Fri Nov 12 13:16:11 BRST 2010." This is version 0.42. The promissed version 0.5 — with if and while statements, which was what I had in mind — did not come out of the paper yet. I don't even know if it will, in fact, but it would be very nice.

Special thanks here goes to Rafael D'Almeida[1] for the changes in this version. It was very nice that he did this in a time when I was using this software the most — studying financial mathematics, competing for government jobs in Brazil. The variable `p` is now a function to the effect of a usual `pop()` function; that is, instead of a mere read-only previous value variable, we now keep a stack of values which each can be popped by a call a mention to p. So, yes, it's a function which need no arguments, nor parenthesis, for its invocation. The old read-only `p` variable is now called `_`.

The text — the presentation of the software — in this document reflects the changes.

[1] `<rafael@kontesti.me>`

1.2 History: changes from 0.4.

The current release date for version 0.41 is “Fri Oct 05 21:49:16 EDT 2007.” This is version 0.41; statements can now break into more than one line. There’s a version 0.5 in progress for a while now which is completely rewritten and much more powerful. More work is necessary, however.

1.3 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 tem here.

The associativity of each is given by `grammar.y`. For example, `^` is right associative; that is, `2^3^4 = 2^(81)` and not `8^4`. The operator `=` is also right associative. But all the others are left. So

$$1 - 2 - 3 = (1 - 2) - 3 = -4$$

and not 2. It indeed makes a difference; subtraction is not associative. Defining an associative direction for addition is not necessary, but we do it anyway so that we don’t clash with `yacc`.

1.4 An overview of some of the features

We may separate statements with a semi colon; much like the C language. This is nice because we can then write

$$x = 1;\ y = x^2 + 67;\ \pi \cdot y^2;$$

instead of separating each statement in a single line. The number π is actually written as `pi`, so is `e` and others like `deg`. 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 read-only variable `_`. This is nice because we can then implement Newton’s Method as follows.

For example, suppose that you 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 $p(-1) = -1$. 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
```

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

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

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

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

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

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

1.5 The Almeida stack

We also — starting at version 0.42 — keep a stack of the `_` values. We call it the Almeida stack in homage to Rafael D'Almeida whom we can thank for the feature and who implemented it very timely.

To pop values from the stack, we just need to mention `p`. So `p` is our pop() function; just observe that we need not use parenthesis to invoke it.

Beware of the obvious nonbackward compatible change of behavior: `p` now controls a stack. Hence, each mention of it implies a different value, and possibly a run-time error if the stack is empty, contrary to `_`, which yields the same previously printed value.

1.6 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 variables can be freely used. You may use a, b, c, d, et cetera. You may also use *this*, *that*, or any other name, but I have placed 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. The strictness here also avoids some memory allocation complications.

1.7 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 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.

1.8 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 the subtraction is convenient.

1.9 List of functions and operators

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