

Re: parser needed

Source: <http://coding.derkeiler.com/Archive/General/comp.programming/2004-01/3083.html>

From: Malcolm (malcolm_at_55bank.freeserve.co.uk)

Date: 01/30/04

Date: Fri, 30 Jan 2004 20:17:03 -0000

"Mahdiarnt" <mahdiarnt@yahoo.com> wrote in message
> *Is there any out there that I can copy in my source file? Or any other*
> *solution?*
>

Feel free to modify this as you wish.

```
#include <assert.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <math.h>
```

```
#include <stdio.h>
```

```
#include "eval.h"
```

```
/*
  evaluates an arithmetical expression.
```

```
+ - * / % (modulus) ()
```

```
also adds keywords
```

```
sin,
cos,
tan,
ln,
pow,
sqrt,
e
PI
```

```
returns the value of the expression
```

```
error – return for error code
```

```
0 = OK
```

```
1 = an error
```

```
*/

/* tokens defined */
#define EOL 0
#define VALUE 1
#define PI 2
#define E 3
#define MOD 4
#define SIN 5
#define COS 6
#define TAN 7
#define LN 8
#define POW 9
#define DIV 10
#define MULT 11
#define OPAREN 12
#define CPAREN 13
#define PLUS 14
#define MINUS 15
#define SHRIEK 16
#define COMMA 17
#define Sqrt 18
#define SPACE 19
#define ERROR 20

static double getvalue(const char *str, int *nread);
static double expr(void);
static double term(void);
static double factor(void);
static void match(int tok);

static int gettoken(const char *str);
static int tokenlen(const char *str, int token);

static double factorial(double x);
static double getvalue(const char *str, int *len);

static const char *string; /* string we are parsing */
static int token; /* current token (lookahead) */
static int errorflag; /* set when error in input encountered */

/*
  evaluate an arithetical expression.
  Params: str – string containing expression
         error – return pointer, 0 on OK, 1 if error
  Returns: the value of the expression
*/
double eval(const char *str, int *error)
{
  double answer;
```

```
errorflag = 0;
string = str;
token = gettoken(str);

answer = expr();

match(EOL);

if(errorflag)
{
    if(error)
        *error = 1;
    answer = 0.0;
}
else
if(error)
    *error = 0;

return answer;
}

/*
parses an expression
*/
static double expr(void)
{
    double left;
    double right;

    left = term();

    while(1)
    {
        switch(token)
        {
        case PLUS:
            match(PLUS);
            right = term();
            left += right;
            break;
        case MINUS:
            match(MINUS);
            right = term();
            left -= right;
            break;
        default:
            return left;
        }
    }
}
```

```
/*
  parses a term
*/
static double term(void)
{
  double left;
  double right;

  left = factor();

  while(1)
  {
    switch(token)
    {
case MULT:
    match(MULT);
    right = factor();
    left *= right;
    break;
case DIV:
    match(DIV);
    right = factor();
    if(right != 0.0)
      left /= right;
    else
      errorflag = 1;
    break;
case MOD:
    match(MOD);
    right = factor();
    left = fmod(left, right);
    break;
default:
    return left;
    }
  }
}

/*
  parses a factor
*/
static double factor(void)
{
  double answer = 0;
  int len;

  switch(token)
  {
    case OPAREN:
      match(OPAREN);
```

```
    answer = expr();
    match(CPAREN);
    break;
case VALUE:
    answer = getvalue(string, &len);
    match(VALUE);
    break;
case MINUS:
    match(MINUS);
    answer = -factor();
    break;
case E:
    answer = exp(1.0);
    match(E);
    break;
case PI:
    answer = acos(0.0) * 2.0;
    match(PI);
    break;
case SIN:
    match(SIN);
    match(OPAREN);
    answer = expr();
    match(CPAREN);
    answer = sin(answer);
    break;
case COS:
    match(COS);
    match(OPAREN);
    answer = expr();
    match(CPAREN);
    answer = cos(answer);
    break;
case TAN:
    match(TAN);
    match(OPAREN);
    answer = expr();
    match(CPAREN);
    answer = tan(answer);
    break;
case LN:
    match(LN);
    match(OPAREN);
    answer = expr();
    match(CPAREN);
    if(answer > 0)
        answer = log(answer);
    else
        errorflag = 1;
    break;
case POW:
```

```
    match(POW);
match(OPAREN);
answer = expr();
match(COMMA);
    answer = pow(answer, expr());
match(CPAREN);
break;
case Sqrt:
    match(Sqrt);
    match(OPAREN);
    answer = expr();
    match(CPAREN);
    if(answer >= 0.0)
        answer = sqrt(answer);
    else
        errorflag = 1;
        break;
default:
    errorflag = 1;
    break;
}

while(token == SHRIEK)
{
    match(SHRIEK);
answer = factorial(answer);
}

return answer;
}

/*
check that we have a token of the passed type
(if not set the errorflag)
Move parser on to next token. Sets token and string.
*/
static void match(int tok)
{
    if(token != tok)
    {
        errorflag = 1;
    }

    while(isspace(*string))
        string++;

    string += tokenlen(string, token);
    token = gettoken(string);
    if(token == ERROR)
        errorflag = 1;
}
}
```

```
/*
  get a token from the string
  Params: str – string to read token from
  Notes: ignores white space between tokens
*/
static int gettoken(const char *str)
{
  while(isspace(*str))
    str++;

  if(isdigit(*str))
    return VALUE;

  switch(*str)
  {
    case 0:
      return EOL;
    case '/':
      return DIV;
    case '*':
      return MULT;
    case '(':
      return OPAREN;
    case ')':
      return CPAREN;
    case '+':
      return PLUS;
    case '-':
      return MINUS;
    case '!':
      return SHRIEK;
    case '%':
      return MOD;
    case 'e':
      return E;
    case ',':
      return COMMA;
    default:
      if(!strncmp(str, "sin", 3))
        return SIN;
        if(!strncmp(str, "cos", 3))
          return COS;
          if(!strncmp(str, "tan", 3))
            return TAN;
            if(!strncmp(str, "ln", 2))
              return LN;
              if(!strncmp(str, "pow", 3))
                return POW;
                if(!strncmp(str, "PI", 2))
                  return PI;
                  if(!strncmp(str, "sqrt", 4))
```

```
return Sqrt;

return ERROR;
}
}

/*
get the length of a token.
Params: str – pointer to the string containing the token
        token – the type of the token read
Returns: length of the token, or 0 for EOL to prevent
         it being read past.
*/
static int tokenlen(const char *str, int token)
{
    int len = 0;

    switch(token)
    {
        case EOL:
            return 0;
        case VALUE:
            getvalue(str, &len);
            return len;
        case PI:
            return 2;
        case E:
            return 1;
        case MOD:
            return 1;
        case SIN:
            return 3;
        case COS:
            return 3;
        case TAN:
            return 3;
        case LN:
            return 2;
        case POW:
            return 3;
        case Sqrt:
            return 4;
        case DIV:
            return 1;
        case MULT:
            return 1;
        case OPAREN:
            return 1;
        case CPAREN:
            return 1;
        case PLUS:

```

```
return 1;
case MINUS:
return 1;
case SHRIEK:
return 1;
case COMMA:
return 1;
case SPACE:
return 1;
case ERROR:
return 0;
default:
assert(0);
return 0;
}
}

/*
compute x!
*/
static double factorial(double x)
{
double answer = 1.0;
double t;

for(t=1;t<=x;t+=1.0)
answer *= t;
return answer;
}

/*
strtod implementation:
Params: str – pointer to string
        nread – return pointer for no bytes read
Returns: value of the string
*/
static double getvalue(const char *str, int *len)
{
char *end;
double answer;

while(isspace(*str))
str++;
answer = strtod(str, &end);
*len = end – str;

return answer;
}

static double oldgetvalue(const char *str, int *nread)
{
```

```
double answer = 0.0;
int ndecimal = 1;

while(isspace(*str))
str++;

assert(isdigit(*str));

*nread = 0;

while(isdigit(*str))
{
    answer *= 10;
    answer += *str - '0';
    *nread += 1;
    str++;
}
if(*str == '.')
{
    str++;
    *nread += 1;

while(isdigit(*str))
{
    answer += (*str - '0') / (10.0 * ndecimal);
    ndecimal++;
    str++;
    *nread += 1;
}
}

return answer;
}
```