AN EXPRESSION EVALUATOR IN FORTRAN

Wilton Pereira da Silva[a]*, Cleide M. D. P. S. Silva[b], Ivomar Brito Soares[c], José Luís do Nascimento[d], Cleiton Diniz Pereira da Silva e Silva[e]

[a, b] *Professors of the Department of Physics*
[c, d] *Electrical Engineering and Scientific Initiation Scholarship Students – PIBIC – CNPq*
Sciences and Technology Center
Federal University of Campina Grande
Aprígio Veloso St., 882, ZIP Code 58109-970, Campina Grande, PB, Brazil
[e] *Master's Degree in Electronic Engineering, Aeronautics Institute of Technology*
Marechal Eduardo Gomes Sq., 50, Vila das Acácias, ZIP Code 12228-900, São José dos Campos, SP, Brazil

Abstract

This paper aims to communicate the development of an expression evaluator (function parser) for the Fortran programming language, to make its source code available and to show the results of a comparative analysis among its performance and two other available ones (the only ones found in successive searches on the internet), in open source, for this language. The developed parser presented a performance significantly superior, in speed, to the similar tested.

Keywords

Expression evaluator; function parser; open source; Fortran.

I - Introduction

Few years ago, professors of the Instrumentation Group for the Teaching of Physics (Grupo de Instrumentação para o Ensino da Física, GIEF) of the Department of Physics (Departamento de Física, DF) of the Sciences and Technology Center (Centro de Ciências e Tecnologia, CCT) at the Federal University of Paraíba (Universidade Federal da Paraíba, UFPB) developed, at the DOS platform, a curve fitting program using the Fortran 77 programming language. The program, at that time called "Ajuste", was designed to fit a function to experimental data using the non-linear regression technique. This program was initially used by students of the Experimental Physics I course offered by DF/CCT/UFPB, and posteriorly by students and researchers of other Brazilian universities. As it is known, the DOS platform has faced a constant decline in its use, and has been systematically substituted by new platforms, more powerful and friendly. Nowadays its use practically doesn't exist and the programs already developed for this platform needed to be rebuilt in these new ones.

*Corresponding author
E-mail addresses: wiltonps@uol.com.br (W. P. Silva and C. M. D. P. S. Silva), ivomarbrito@uol.com.br (I. B. Soares), jluisn@dee.ufcg.edu.br (J. L. Nascimento), cleitondiniz@directnet.com.br (C. D. P. S. Silva)

In face of this new panorama, the developers of "Ajuste" decided to develop a new version of this program for the Windows platform and it was named "LAB Fit Curve Fitting Software" (LABFit) [9]. As it's known, a curve fitting software normally has a library with a set of pre-defined functions and an option for the users to be able to write their own fitting function, which requires a specific code called "expression evaluator" (parser) for the calculation of the function for a given set of values of the independent variables. Beyond the study of new commands of Fortran 90 [3] and of the graphical part ([5] and [10]), during the migration process there was the need to solve one problem: how to obtain an expression evaluator, in open source, developed in Fortran, whose implementation was adequate to the necessities of the program in development. In reality, repetitive searches on the internet ended in only one evaluator coded in Fortran, which has been developed in Australia [6]. But the "Australian parser ", although it was of easy incorporation into the program and useful in many applications, was prohibitively slow for the required purposes: non-linear regression. During the search for a result in a curve fitting program, the convergence process may take hundreds of thousands of iterations and it requires an extremely efficient and optimized code for the evaluation of parameters of the fitting function.

The DOS version of the software didn't need an expression evaluator because the source code was made of parts and one of them was the specified user function. These files were put together through a lot file and then compiled via a Fortran compiler [13], which was part of the package. Naturally, this type of resource doesn't exist in Windows due to the complexity and the size of the compilers for this platform, and also due to problems related to limitations concerning the right of use. Therefore, the natural solution to solve such impediment would be the utilization of a parser, but the only one available for Fortran was not considered adequate to the goals of the program in development. Hence, the only alternative was to develop an expression evaluator for the LABFit.

II – Requirements of the Expression Evaluator

In a curve fitting procedure, using the non-linear regression technique, there is an interactive process where an optimum point is obtained through a convergence condition. Such process can involve thousands of iterations, which takes time. This requires that the code for the evaluator of a given expression be efficient and optimized. For the specific case of regression, a parser must attend two requirements. The first one is that,

once read the string with the fitting function, the building process of the corresponding arithmetical expression must be done only once, right after the reading of the string. This guarantees great time savings in an iterative process. The second requirement is that, once the expression to be evaluated was interpreted and built, the calculation of such expression for a set of variable values be done in an optimized way, which guarantees the needed speed in a process that may be repeated hundreds of thousands of times.

III – Evaluating Expressions – General Notions

The procedure in which a program receives a string containing a numerical expression such as, for example, (4*3)/10^2, and returns the appropriate answer is generically called expression evaluation. Such procedure is the base of all compilers and language interpreters, of mathematical programs and of everything that needs to interpret mathematical expressions in a way that the computer can handle them.

The algorithm used for the development of the parser was the recursive descent. Such algorithm is described in several books about the C language (for example [7]), and was adapted to Fortran. As the objective of this paper is just to communicate the development of a parser coded in Fortran and to make available its source code (and not to discuss the algorithm itself), it will only be shown the basic ideas of the performed study.

III.1 – Elements of an Expression - Precedence

For the purposes of the development of a parser, it should initially be stated that mathematical expressions are formed by the following items:

- Numbers;
- Operators: + addition, - subtraction, / division, * multiplication, ** or ^ exponentiation;
- Brackets;
- Functions: sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, sind, cosd, tand, log, log10, nint, anint, aint, exp, sqrt, abs, floor;
- Variables.

All the previous items can be combined, obeying the algebra rules, to form mathematical expressions. Next, there are some examples:

```
1/(a+b*x**(c-1) + 4.321)
a+(b-a)/(1+exp(-c*(x-d)))
a+b*log(x1)+c*log(x1)**2+d*log(x2)+e*log(x2)**2
(x+y+z+x*y+x*z+y*z+x/y+x/z+y/z+x*cos(x)+y*sin(y)+z*tan(z)*2/(x+y+z+x*y+x*z+y*z+x/y+x/z+y/z
+x*cos(x)+y*sin(y)+z*tan(z))*3+sqrt(x*y*z+x+y+z)*log10(sqrt(x*2+y*2+z*2)+x+y+z))
```

At the development of a parser, an important concept is the operators and functions precedence [1]. Such concept states which operation must be performed first and, therefore, defines the sequence in which the operations must be executed with the purpose to obtain a correct evaluation of the interpreted expression. For the purposes of this evaluator it was assumed the precedence presented in table 1.

[table 1]

As an example, consider the evaluation of the following expression:

16 - 3 * 4.

Naturally, this expression has as result the value 4. Although it seems to be an easy task to create a source code that calculates the final value for this specific string, the question that must be raised is the development of a code that gives the correct answer for any arbitrary expression. The operators can not simply always be taken in a left to right order. The problem becomes even more complex when brackets, exponentiation, variables and functions are added to the expression.

III.2 – Analysis of an Expression

There are several possibilities to analyze and build a mathematical expression. In the case of a recursive descent parser, the expressions are imagined as being recursive structures, that means, such expressions are defined in terms of themselves. Just to have a basic notion about the rules and the ideas used in the development of an evaluator; imagine an expression containing the following elements: +, -, *, /. In this case, the expression can be defined, from the reading of a string, with the utilization of the following basic rules [4]:

Expression → term [+term][-term];

Term → factor [*factor][/factor];

Factor → variable, number or (expression).

At the terminology previously shown, the square brackets designate an optional element and the symbol "→" means "produces". Such rules are normally called "rules of expression production". This way, it is possible to interpret the definition of a term as: "term produces factor multiplied by factor or factor divided by factor". The operators' precedence is implicit in the way how an expression is written. A parser must identify the priorities in the sequence of the operations and also must execute them in the identified sequence.

To illustrate the utilization of the rules presented above, take the expression 6+3*D, in which it is possible to identify two terms: the first one is the number 6 and the second one is the product given by 3*D. The second term has two factors: 3 and D. As it is seen, these two factors are a number and a variable.

The presented rules are the essence of a recursive descent analyzer, that is basically a set of functions mutually recursive that operate in a linked way. At each step, the analyzer executes the specified operations in the algebraically correct sequence. In order to have a concrete idea of how this process works, analyze the following expression:

9/3 – (100 + 56)

- Initially the first term, 9/3, is taken. Then, each factor is taken and the integer division is executed. The result value is 3;
- The second term, (100+56) is taken. At this point, the second sub-expression is analyzed recursively. The two factors are taken and, then, added. The result value is 156;
- It is returned from the recursive call and then 156 is subtracted of 3. The answer is -153.

There are two basic points to be remembered about this recursive vision of the expressions. First, the precedence of the operators is implicit in the way how the production rules are defined. Second, this method of analysis and building of expressions is quite similar to the way we humans evaluate mathematical expressions.

III.3 – Parse Tree

To illustrate the presented notions it is shown at figure 1 a parse tree referring to the recursive descent process for the expression: – A + 5 * B / (B – 1) [2].

[Figure 1]

IV – The developed Source Code

The source code was developed to attend the pre-defined basic requirements in section II for the specific needs of the LABFit, according to the rules that were established in section III. The final result is available at the web site indicated at the Ref. [12].

Basically, the code is constituted of the following parts: 1) the parser itself comprised in a module called interpreter.f90, 2) the main program, called test.f90, containing some examples. Once the main program has the information about the string, it calls a subroutine of the module interpreter.f90, called “init”, whose function is to make the interpretation, that means, the building of the mathematical expression. With this task accomplished, the function that evaluates the expression, called “evaluate”, is evoked and the value of the expression is returned for a set of variable values. The process of evaluating an expression can be repeated for other set of variables, without the need of rebuilding the expression, for how many times it may be necessary. It is also possible, in the same execution, to change the string and to repeat the interpretation and evaluation processes for the new specified expression. To do so, it is only needed to call the subroutine named “destroyfunc”, before informing the new string and repeating the process previously described.

The philosophy used at the development of this parser was the same normally used in the development of modules, in a general way. This means that the Fortran programmers, which will use the developed parser in their programs, don’t have to know its source code. They just need to add the module interpreter.f90 in their projects, define the string and call the subroutine “init”, followed by the calling of the function “evaluate” as it is shown in the example program test.f90.

V – Comparative Analysis

During the coding process of the parser its developers got known, through the internet, of another similar one developed in Germany [8], just finished. The performance of this parser was tested by adding it to

the LABFit and the results were considered good. Moreover, it was performed a set of tests to compare the performance of the developed evaluator and of the two other ones ([6] and [8]). The tests results would indicate the appropriated parser to be used at the curve fitting software.

The tests consisted on the realization of five million iterations for twenty five distinct expressions, measuring the time that each evaluator took to perform such task. The tests were performed in a computer Intel™ Pentium III, 128 Mbytes of RAM memory and the compilation was done at the Compaq Visual Fortran (CVF 6.5) studio, using the QuickWin Application option. The elapsed time for the direct calculation, which takes the smallest execution time, of each compiled expression was also measured in order to have an idea of the slowness of a Fortran coded parser execution. The performance of the evaluators was analyzed only in terms of the elapsed time for the tasks execution, because the numerical results of all of them were equivalent and compatible with the numerical result of the compiled expression. The complete report for all the expressions was made available on the internet [11]. Next, it will be shown the obtained results for five, among the twenty five, tested expressions.

1st Expression:

(x+y+z+x*y+x*z+y*z+x/y+x/z+y/z+x*cos(x)+y*sin(y)+z*tan(z)*2/(x+y+z+x*y+x*z+y*z+x/y+x/z+y/z+x*cos(x)+y*sin(y)+z*tan(z))*3+sqrt(x*y*z+x+y+z)*log10(sqrt(x*2+y*2+z*2)+x+y+z))

For x = 0.175, y = 0.110 and z = 0.900:

Final Result: 5.481916

Compiler direct calculation time: 9.644s

Developed parser time (LABFit): 48.119s

“Australian parser” time (Stuart Midgley): 178.060s

“German parser” time (Roland Schmehl): 77.681s

2nd Expression:

a+b*x1

For a = 0.900, b = 0.100 e x1 = 0.508:

Final Result: 0.9508000

Compiler direct calculation time: 0.080s

Developed parser time (LABFit): 1.872s

“Australian parser” time (Stuart Midgley): 22.242s

“German parser” time (Roland Schmehl): 3.205s

3rd Expression:

cosh(log(abs(y*z+x**2+x1**x2)))+a*d*(exp(c*f)+154.3)

For x = 0.175, y = 0.110, z = 0.900, a = 0.900, c = 0.110, d = 0.120, f = 0.140, x1 = 0.508 and x2 = 30.000:

Final Result: 20.69617

Compiler direct calculation time: 7.150s

Developed parser time (LABFit): 17.114s

“Australian parser” time (Stuart Midgley): 59.876s

“German parser” time (Roland Schmehl): 23.664s

4th Expression:

atan(sinh(log(abs(exp(z/x)*sqrt(y+a**c+f*e)))))

For x = 0.175, y = 0.110, z = 0.900, a = 0.900, c = 0.110, f = 0.140 and e = 0.130:

Final Result: 1.559742

Compiler direct calculation time: 9.533s

Developed parser time (LABFit): 15.573s

“Australian parser” time (Stuart Midgley): 53.958s

“German parser” time (Roland Schmehl): 20.359s

5th Expression:

atan(sinh(log(abs(exp(z/x)*sqrt(y+a**c+f*e)))))*cos(log(abs(sqrt(y+a**c+f*e))))

For x = 0.175, y = 0.110, z = 0.900, a = 0.900, c = 0.110, f = 0.140 and e = 0.130:

Final Result: 1.557368

Compiler direct calculation time: 14.258s

Developed parser time (LABFit): 24.518s

"Australian parser" time (Stuart Midgley): 76.528s

"German parser" time (Roland Schmehl): 32.915s

VI – Conclusions

From the analysis of the data presented in section V, it is possible to notice that the developed expression evaluator showed, for the desirable requirements, a performance superior than the other two existing ones, in all performed tests. In this way, the parser was not only used at the software LABFit, but is also being made available on the internet for the free utilization by the interested ones. The parser developers believe, along with the others ones involved in similar projects, in having helped to fill a gap in the libraries of Fortran, which is a language still quite used nowadays by engineers and scientists.

References

[1] - A. V. Aho, R. Sethi, J. D. Ulman, *Compilers – Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, pp. 203 – 208, 1987.

[2] - F. L Bauer, F. L. De Remer, A. P. Ershov, D. Gries, M. Griffiths, U. Hill, J. J. Horning, C. H. A. Koster, W. M. Mckeeman, P. C. Poole, W. M. Waite, *Compiler Construction – An Advanced Course*, Springer-Verlag, 2nd Edition, p. 6, 1976.

[3] - S. J. Chapman, *Fortran 90/95 for Scientists and Engineers*, WCB/McGraw-Hill, 1st Edition, 1998.

[4] - C. N. Fischer and R. J. LeBlanc Jr., *Crafting a Compiler With C*, The Benjamin/Cummings Publishing Company, pp. 382 – 387, 1991.

[5] - N. Laurence, *Compaq Visual Fortran – A Guide to Creating Windows Applications*, Digital Press, Woburn MA, USA, 2002.

[6] - S. Midgley, Department of Physics, University of Western Australia, "*Australian Parser"*, online available on the internet at http://www.netspace.net.au/~smidgley/fortran/.

[7] - H. Schildt, *C Completo e Total*, Makron Books, 3rd edition, pp. 584 – 606, 1996.

[8] - R. Schmehl, "*German Parser"*, online, available on the internet at http://www.its.uni-karlsruhe.de/~schmehl/functionparserE.html.

[9] - W. P. Silva, C. M. D. P. S. Silva, *LAB Fit Curve Fitting Software*, online, available on the internet at http://www.angelfire.com/rnb/labfit/.

[10] - W. P. Silva et al, *VFortran Tutorial*, online, available on the internet at http://www.extension.hpg.com.br/.

[11] - I. B. Soares, J. L. Nascimento, W. P. Silva, *Full Report of the Performed Tests*, online, available on the Internet at http://www.extension.hpg.com.br/.

[12] - I. B. Soares, J. L. Nascimento, W. P. Silva, *Parser Source Code,* online, available on the internet at http://www.extension.hpg.com.br/.

[13] - University of Waterloo, WATFOR77.EXE V 3.1, August 1989.

Figure Legends

Fig. 1. Parse tree for the expression – A + 5 * B / (B – 1).

Tables

Higher Precedence	Brackets
	Functions
	** or ^
	*, /
Lower Precedence	+, -

Tab. 1. Operators Precedence

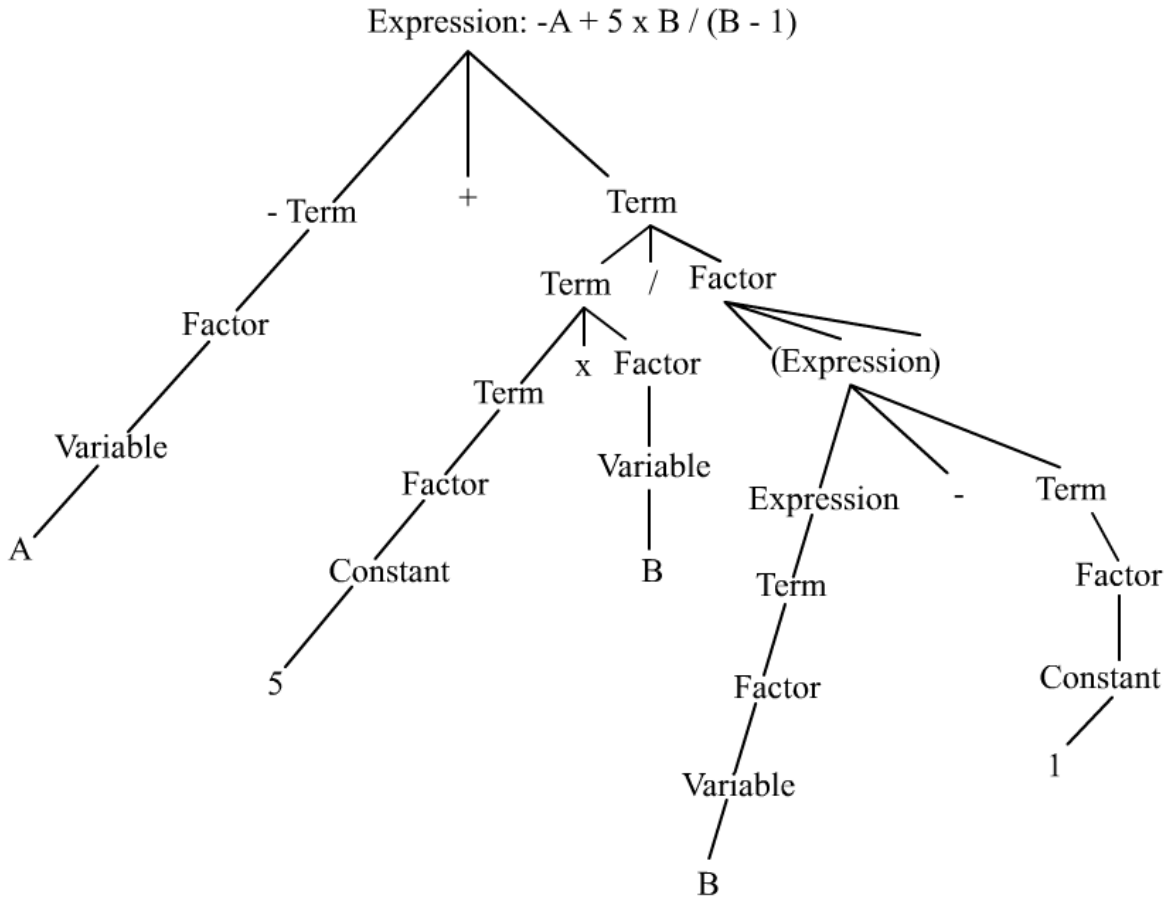
Expression: -A + 5 x B / (B - 1)
- Term
+
Term
Term
/
Factor
Factor
x
Factor
(Expression)
Term
Variable
Factor
Variable
Expression
-
Term
A
Constant
B
Term
Factor
5
Factor
Constant
Variable
1
B

Figure 1