SYNTACTIC PARSING AND PLOTTING OF MATHEMATICAL EXPRESSIONS

INFORMATICA 1/89

Descriptors: SYNTAX, ANALYSIS, TEXT NATURAL, SOFTWARE, TREE GRAMMAR, MATHEMATICAL ANALYSIS, LANGUAGE ANALYSIS

Nikola Bogunović, Institut R. Bošković
Zagreb

The paper presents the parsing problem of a simple context free language. The "language" sentences are mathematical expressions with one variable. A computer program parses the expression according to the developed context free grammar rules. Upon building a parse tree, the program evaluates the expression over a given range of variable values, and plots the result on the screen. Even though the program is developed on a PC AT personal computer, it is highly portable since the C programming language is used, and graphics hardware dependent routines are removed in a separate module.

SINTAKTIČKA ANALIZA I GRAFIČKI PRIKAZ MATEMATIČKIH IZRAZA

U radu je predstavljen problem analize rečeničnog sloga u kontekstno slobodnom jeziku. "Jezičnu rečenicu" predstavlja matematički izraz s jednom varijablom. Računarski program razlaže izraz u skladu s razvijenim kontekstno slobodnim gramatičkim pravilima. Program gradi stablo sastavnih dijelova matematičkog izraza, nalazi vrijednosti izraza za dati niz vrijednosti varijable, i grafički prikazuje rezultat. Iako je program razvijen na računalu PC AT, jednostavno je prenosiv na druga računala, jer je pisan u višem jeziku (C), a grafičke, sklopovski ovisne rutine premještene su u odvojen programski modul.

INTRODUCTION

Language parsing has traditionally been one of the most intriguing research areas of artificial intelligence. The problem here is to take the information provided from the outside world and translate it into a precise internal representation. By the internal representation we mean a semantic representation, a common data structure produced or operated on by various program modules. Even though, a common data structure of internal representation may have different forms, it is assumed that the translations from one to another is easy, and all forms are variants of the same abstract representation.

The application of language parsing, in the context of this paper, is directed to engineering problems, e.g. intelligent industrial process control, rather than attempt to solve an over aspiring and not very well defined problem like automatic language translation. It is more sensible to work on the internal representation generation, since this is an intermediate point between words and actions.

The problem of language parsing can be divided into three areas, with the apparent ambiguity at each level [1]:

1. acoustic-phonetic: time domain and frequency domain analysis of the incoming sound, and translation the input into words.
2. morphological-syntactic: taking words and establishing the syntactic form of the utterance.
3. semantic-pragmatic: finding out the meaning from the syntactically analyzed utterance.

In this paper we will concentrate on the problems of second and third level only. Our goal is to develop an internal representation from the correctly received input stream of information, looking at the major data structures the computer program uses. The problems at first level, and partially at second, can be bound loosely to speech recognition, with major research advances and results given in [2] and [3].

A computer program that translates from any natural language to internal representation, must in the first step syntactically analyze, or parse, the sentence. In the process, one needs to know the rules of syntax for the language, specifying the legal syntactic structures for a sentence.

A parsed sentence is usually presented in a tree structure known as parse tree or phrase marker. Specifying the structure of a sentence in a formal knowledge structure, by a series of production rules i.e. grammar, could be far too complex for any natural language [4]. The grammar must be, in that case, context-free, indicating how to replace a nonterminal node of the parse tree with lower constituents, without any reference to the context in which the nonterminal node finds itself [5].

Since the question, whether a natural language is context-free or not, is yet to be answered and a context-free grammar for such a language devised, in this paper we will concentrate on a much easier task of parsing the arithmetic expressions, which are inherently controlled by a context-free grammar. Nevertheless, the presented parser program applies the very same methodology of natural language parsing, and develops a parse tree of an arithmetic expression of considerable complexity.

A rough outline of context-free parsing can also be found in [6]. This paper extends the presented idea with different and more efficient algorithms and data structures available in C language, introduces complex expressions based on a family of new functions, evaluates the function of one variable and develops a graphics interface for the presentation of the evaluated function over a given range of values on the terminal of a PC XT/AT or PS/2 personal computer. The computer program was written in Microsoft C, and linked with an assembly language program to run industry standard monochrome or colour graphics routines.

THE SYNTAX OF A LANGUAGE

A context-free language is one whose syntax can be specified by a set of rules, usually called productions, or simply grammar. A context- free derivation of a sentence always start with a nonterminal node of the parser tree, or a nonterminal constituent, i.e. with a node that appears only in the interior of the tree structure, and not in the final sentence. Each nonterminal node is then replaced by the right hand side of the rule, until we have only terminal nodes, or terminal constituents, i.e. nodes that appear in the final sentence.

A very simple context-free grammar might have the following rules:

```
sentence(s) --> noun-phrase(np) verb-phrase(vp)
verb-phrase(vp) --> verb(v) noun-phrase(np)
verb-phrase(vp) --> verb(v)
noun-phrase(np) --> determiner(d) noun(n)
noun-phrase(np) --> proper-noun(prn)
```

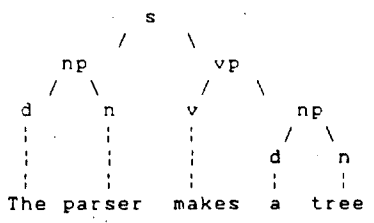
The last two rule pairs may be combined with the logical operator OR (¦):

```
vp  -->  v np  ¦  v
np  -->  d n   ¦  prn
```

The syntax of grammar rules can be also described by the recursive set of grammar rules themselves:

```
grammar_rule --> grammar_head [-->] grammar_body
grammar_head --> nonterminal_node
grammar-head --> nonterminal_node terminal_node
grammar_body --> grammar_body grammar_body
grammar_body --> grammar_item
grammar_item --> nonterminal_node
grammar_item --> terminal_node
```

The application of the context-free grammar to a simple sentence like "The parser makes a tree." is exceedingly clear from the following parse tree:



This simple example creates more questions than it really indicates the solution of the problem. It is extremely difficult to express every rule of grammar with context-free rules. The problem of number agreement (singular-plural), morphology (differences between go, goes, going), reflexivization (reflexive pronouns), imperatives, passive case, etc. are just the few most common. One can add weakly context-free structure in the variation of the above example with the sentence like "It does."

Programming languages, on the other hand, are context-free, and principal compiler task for such a language is to parse it. In that case we may even talk about the efficiency of parsing, optimal parsing, and so forth. The expansion of RISC computer architecture, as a contemporary industry trend, causes the compiler construction and language parsing problem to be of outmost importance.

Mathematical expressions are the most simple case of finite character strings whose syntax can be captured in a context-free grammar. Since our primary goal in this paper is to show the principles of a working parser, we have constrained the presented application to a "language" that can be described with only a few production rules.

Let us assume a mathematical equation with one variable, floating point constants, four arithmetic (binary) operators, and five unary operators, with left to right evaluation in the traditional fashion. An example of such an expression is:

```
f(x)=2.5+3.4-4.5sin(cos(5.8x+2.2))          (1)
```

We have decided upon these basic set of operations with the following order of precedence:

```
    -  :  unary minus
sin(x) :  sine function
cos(x) :  cosine function
log(x) :  natural log
exp(x) :  exponential
    *  :  multiplication
    \  :  division
    +  :  addition
    -  :  subtraction
```

The syntax of the expression (1) can be captured in the recursive set of context-free grammar rules. We may use the notation introduced at the beginning of this paper or instead, we may use the familiar and traditional Backus-Naur form, from the computer science literature:

```
<expr> ::= <term> | <term>+<expr> |
           <term>-<expr>
<term> ::= <factor> | <factor>*<term> |
           <factor>\<term>
<factor> ::=  <variable> | <number> |
              -<factor> | sin<expr> |
              cos<expr> | log<expr> |
              exp<expr> | (expr)
```

It is evident that the functions sin, cos, log, and exp are implemented as unary functions, like unary minus. Implied multiplication, used in the input expression, is later changed to explicit (*).

The application of these production rules to the equation (1), is presented in Fig.1. Parsing starts with the <expr>, which according to the first rule of our grammar may have three forms: a <term>, <term>+<expr> or a <term>-<expr>. Since it is obviously a <term>+<expr>, further application of grammar rules to <term> part yields a <factor>, then a <number> which is a floating point constant.

Parsing the other part needs a recursive application of the same rules. Since it is an <expr>, we apply the first rule again, which yields <term>-<expr>. The <term> part is a <factor>, a <number> and a constant. The <expr> part is a <term>, which is a <factor>*<term>. The process proceeds until the terminus of all branches is reached, yielding a <number> or a <variable>.

Once a parse tree is constructed, the expression may be evaluated starting at the top of the tree by recursively evaluating left and right branches, and then performing addition or subtraction at the top.

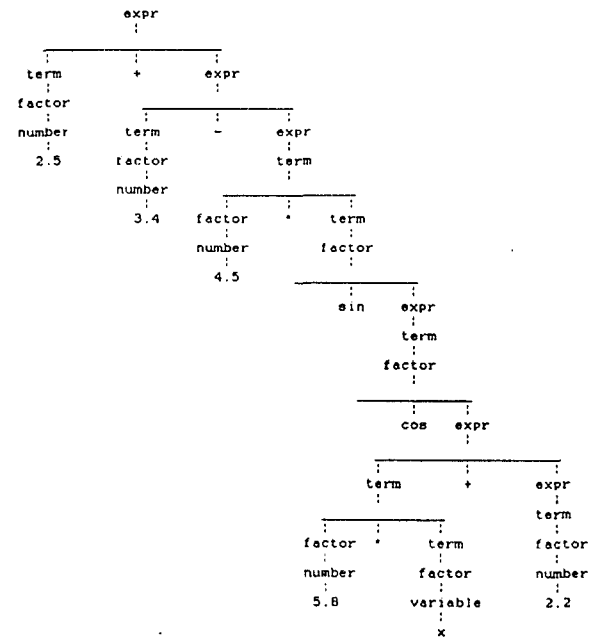


Fig.1 The parse tree of equation (1).

DATA STRUCTURES AND ALGORITHMS

The principal elements of a parse tree are nodes. Looking at Fig.1, we may deduce that there are four kinds of nodes. A node is either a number (a floating point constant), a variable (x), a unary operator node (-,sin,cos,log,exp), or a binary operator node (+,-,*,/). In our case the root node is a binary operator (+) with left and right operands, i.e. <term>+<expr>, according to the first grammar rule. Left operand will be parsed into <factor> --> <number> --> 2.5. Right operand will be parsed recursively starting with the first grammar rule again. All four kinds of nodes can be captured in a structure, in C language sense (7), with the following components:

```
struct node {
        int tag;
        char operator;
        float number;
        struct node *left_operand;
        struct node *right_operand;
             } TREENODE, *TREEPTR;
```

Integer tag identifies a node as a number (tag=0), a variable (tag=1), a binary operator (tag=2), or a unary operator (tag=3). Character operator identifies operator as +, -, *, /, sin, cos, log, or exp. If the node is a number, the floating point value is held in the "number" structure member. If the node is a binary operator, pointers left_operand and right_operand point to the left and right "child" nodes (structures). If the node is a unary operator, only left_operand pointer is used. It is absolutely important to note that the root node structure embeds the whole parse tree, because left and right operands, as the structure members, are pointers to the structures of the same type as the root node itself. This recursive declaration of a node is correct, as given in [7]. Typedef TREENODE and TREEPTR, define a node type structure and a pointer to such a node type structure.

At the beginning of the program, the string, which corresponds to the input equation, is subject to the preprocessing operation. The string is converted to lower case, and all surplus spaces are removed. Since sin, cos, log, and exp functions are implemented as unary operators, they are stripped to a single unique character operators (s,c,l,e). Finally, implicit multiplication is changed to explicit. After the preprocessing phase, our equation (1) would fill an array of characters that would look like:

```
2.5+3.4-4.5*s(c(5.8*x+2.2))          (2)
```

In the next step a parse tree is constructed. Any expression, if not constrained with parenthesis to a subexpression, must start with a term, which must start with a factor (number,variable or unary operator applied to <expr>). In the process of building a parse tree, we actually make the instances of node structures defined earlier. As already stated, the root node contain pointers to the neighbouring structures, and in essence embeds the whole tree. There is an initial function expr(), which calls the function term(), which finally calls function factor(). These functions mirror our grammar rules. The function factor() analyses the beginning of the string. In our case it will find a number 2.5 (a constant), and it will make a number node and return a pointer to its caller. The callee, function term(), will further analyze the string to find if there is a multiplication (*) or a division (/) sign according to the second grammar rule. If not, which happens in our case, term() will return a pointer of a number node to expr(). The expr() function will analyze the string further and find an addition sign, hence the root node is a binary operator node with left operator already established (previously found number node). The right node will be found by a recursive call to expr() function again.

To illustrate the creation of the number node structure, we give the function numbernode(), which is called by the factor() after it has extracted the number and its value from the beginning of the string.

```
#define new()
    (TREPTR) calloc(1,sizeof(TREENODE))

/* This is a global creation of the space
which will hold a node, and return a
pointer to that space. */

#define NULL 0

TREEPTR numbernode(value)
/* take the number value and return a
pointer to a structure */
float value;
/* the type of passed parameter */
{ TREEPTR n;
/* declaration of the pointer */
        n = new();
/* creation of an empty struct. */
        n->tag = 0;
/* it is a number node */
        n->number = value;
/* fill in the value */
        n->left_operand = NULL;
/* numbernode has no neighbours */
        n->right_operand = NULL;
return(n);
/* returning a pointer */
}
```

Since this paper describes the equation parser and plotter, we have included in List 1, an evaluation function which, for a given variable value, will traverse through the parse tree in a recursive search fashion, finding the value of the whole expression. The function eval(root_node_pointer,x) will test the tag of the root node and act accordingly. If the node is a unary or binary operator node, eval() will call itself with new pointers.

```
float eval(n,x)
TREEPTR n;              /* 1st passed parameter is a pointer
                           to the root node structure */
float x;                /* 2nd parameter is a variable value */
{       float op1,op2;
        switch (n->tag)
        { case 0:               /* it is a number node */
                return(n->number);
                break;
          case 1:               /* it is a variable node */
                return(x);
                break;
          case 2:       /* it is a binary operator node */
                op1 = eval(n->left_operand,x);
                op2 = eval(n->right_operand,x);
                      switch(n->operator)
                      {  case "+":
                                return(op1+op2);
                                break;
                         case "-":
                                return(op1-op2);
                                break;
                         case "*":
                                return(op1*op2);
                                break;
                         case "/":
                                return(op1/op2);
                                break; }
           case 3:          /* it is a unary operator node */
                switch(n->operator)
                {       case '-':
                                return(-eval(n->left_operand,x));
                                break;
                        case 's':
                                return(sin(eval(n->left_operand,x)));
                                break;
                        case 'c':
                                return(cos(eval(n->left_operand,x)));
                                break;
                        case 'e':
                                return(exp(eval(n->left_operand,x)));
                                break;
                        case 'l':
                                return(log(eval(n->left_operand,x)));
                                break;
                }
        }
}
```

List 1. Evaluation of the expression.

The presented function eval() is only a basic skeleton of the implemented function, because one has to take great care how binary and unary functions are defined (no negative values for log, divide with zero, etc.), and whether a parenthesis is encountered indicating a subexpression.

Finally, after obtaining domain and value points of the equation, we can display it graphically. The graphics functions greatly depend on the used hardware and can not be given generally. However, since industry standards like personal computers PC XT/AT and PS/2 are readily available, we will show the principles of implementation some simple graphics procedures for these computers. Even within the PC and PS family of computers, graphics standards vary from 320x200 pixels to an impressive 1024x768 pixels (with additional advanced display adapter). In this paper we have chosen to show Hercules monochrome graphics implementation, in belief to represent a popular, yet fully acceptable medium resolution (720x348) graphics standard.

Hercules graphics functions library is a set of memory resident routines, set up by INT10.COM, a program supplied and copywrited by the vendor [8]. We have chosen to implement graphics routines in the assembly language and link them with the main C program to achieve maximum portability. The assembly language program treats Hercules graphics functions as the extension of the standard display control software interrupt procedures (int 10h). All parameters are simply loaded in registers, with the function code in AH register, prior to int 10h call. Unlike the original set of functions within int 10h group, only segment registers are preserved, along with all registers used to pass parameters.

An example of assembly language function, which moves the cursor to the x,y position (move(x,y)), is given below. The caller from the C program will leave x and y coordinates, as parameters, on the stack. It was assumed that C program will run on a PC XT/AT in the small model ("Microsoft" restriction to 64K byte), hence near procedure type.

```
_text segment byte public 'code'
      assume cs:_text
; definitions as required by Microsoft C
      public _move
_move   proc    near
        push    bp
        mov     bp,sp
        push    di,
        mov     di,[bp+4]
; get x from stack
        mov     bp,[bp+6]
; get y from stack
        mov     ah,40h
; it is function "move"
        int     10h
; call function
        pop     di
        pop     bp
        ret
_move   endp
_text   ends
        end
```

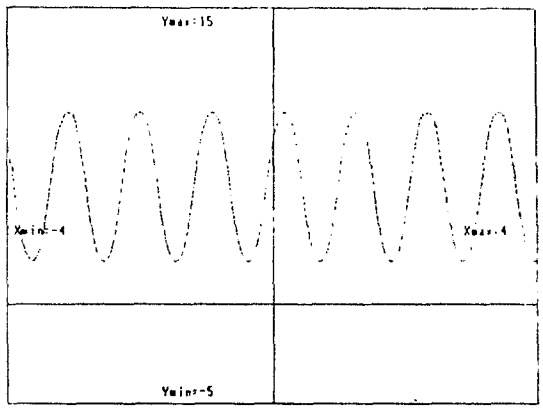


Fig.2 The graph of equation (1).

CONCLUSIONS AND REMARKS

We have studied string parsing techniques, applied to the simple mathematical equations. The syntax of these strings can be described by an elementary context free grammar. Nevertheless, the same principles apply to a broad range of languages described by context free grammars.

To illustrate the parsing, evaluating and plotting operations of the working program, we have presented the graph of the equation (1) in Fig.2. The function is plotted over a domain range (-4,+4). The scale of ordinate values is appropriately chosen to display points between -5 and +15. The program prompts for the scale before it plots the function. By changing the coordinate scale one can easily zoom, scroll and pan the graph, sustaining the same resolution.

It is worth noting that the program embeds implicit precedence rules (multiplication and division before addition and subtraction), and enforced precedence by parentheses, according to the given grammar rules.

REFERENCES:

1. E.Charniak, D.McDermott, Introduction to Artificial Intelligence, Addison-Wesley, Reading, Mass., 1985.

2. J.L.Flanagan, Speech Analysis, Synthesis, and Perceptions, Springer Ferlag, New York, 1972.

3. S.E.Levinson, L.R.Rabiner, M.M.Sondhi, An Introduction to the Application of the Theory of Probabilistic Functions of a Markov Process to Automatic Speech Recognition, Bell Syst. Tech. J., Vol. 62, No. 4, 1982.

4. N.Chomsky, Syntactic Structures, Mouton, The Hague, 1957.

5. A.V.Aho, J.D.Ullman, The Theory of Parsing, Translation and Compiling, Prentice-Hall, Englewood Cliffs, N.J., 1972.

6. J.Amsterdam, Context-free parsing of Arithmetic Expressions, Byte, Vol. 10, No. 8, August 1985.

7. B.W.Kernighan, D.M.Ritchie, The C Programming Language, Prentice-Hall, Englewood Cliffs, N.J., 1978.

8. GRAPHX V1.1 Manual, Hercules Computer Technologies, 2550 Ninth Street, Berkeley, CA 94710, USA.