SYMBOLIC CALCULUS THROUGH PROLOG

Dissertation submitted to the Jawaharlal Nehru University
in partial fulfilment of the requirements
for the award of the degree of

MASTER OF PHILOSOPHY
(Computer Science)

B. BALACHANDRA RAO

SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110067
1987

CERTIFICATE

This work embodied in the dissertation titled, " Symbolic Calculus through Prolog ", has been carried out by Mr. B.BALACHANDRA RAO, a bonafide student of School of Computer & Systems Sciences, Jawaharlal Nehru University, New Delhi - 67.

This work is original and has not been submitted for any other degree or diploma of any other University.

Sri.C.RAVI SHANKAR
Systems Specialist
R & D Centre,
CMC Ltd.
115 SD Road
SECUNDERABAD - 3

Dr.K.K.BHARADWAJ
Associate Professor
School of Computer &
Systems Sciences
Jawaharlal Nehru University
NEW DELHI - 67

Prof.K.K.NAMBIAR
Dean, School of Computer &
Systems Sciences
Jawaharlal Nehru University
NEW DELHI - 67

CONTENTS

ACKNOWLEDGEMENTS

I convey my sincere thanks to my guide Dr.K.K.Bharadwaj, Assoc. Prof., SC & SS, JNU for allowing me to work on this project and for his constant encouragement & valuable guidance throughout the course of this project.

This project would never have been completed without the help of so many people to whom I am deeply indebted.

First I would like to thank Mr.S.Kapoor, Systems Manager, CMC Ltd, Sec'bad for giving me an opportunity to do my dissertation work at CMC Ltd.

Words reflect poorly my gratitude to Sri.C.Ravishankar, Systems Specialist, CMC Ltd, Sec'bad for initiating me into this project and his help throughout the project was warm and unstinting.

It has been a great pleasure working with Mr.Sethuraman. His guidance especially at quelling moments of uneasiness was invaluable.

I thank Mr.U.Bhaskar for his valuable guidance at initial stages of this project.

Lastly I convey my sincere appreciation to all SUNRAY USERS for their abundant patience & tolerence, while I was "Prologing".

I would also like to acknowledge Prof.P.C.P.Bhatt, Head, Dept of Computer Science, IIT Delhi & Prof.K.K.Nambiar, Dean, SC & SS, JNU for taking interest in me and guiding me to CMC Ltd, Sec'bad.

(B.Balachandra Rao)

ABSTRACT

In this report application of Prolog in Symbolic Calculus is discussed with special reference to integration and differentiation. Prolog has been becoming more popular as Artificial Intelligence applications such as expert systems, symbolic computation, natural language processing, natural language interfaces to databases, deductive databases and automatic programming.

Symbolic Calculus program solves elementary symbolic problems, that is, indefinite integrals and differentiation problems. The domain of symbolic integration is restricted to few types of problems - like standard integrals, constant and a function, sum and difference of integrals, integration by parts and some other integrals. Most of the design effort has been spent on integration by parts. Program can be subdivided into two modules, one is the integration & differentiation, which determines the integal/derivative of a given expression and other is the simplification module, which simplifies the integarated/differentiated expression. Symbolic Calculus program was written in Prolog and runs interpretively on the SUNRAY System at R & D Centre, CMC Ltd., Secunderabad.

1. INTRODUCTION

Programs for solving elementary symbolic integration were written in 1960s as AI applications. The first major program was James Slagle's SAINT (Symbolic Automatic INTegrator) written as a 1961 doctoral dissertation at MIT. The program solves elementary symbolic integration problems - mainly indefinite integrals at about the level of a good college freshman. SAINT was written in LISP and run interpretively on the IBM 7090 computer. A second important symbolic integrator program, SIN (Symbolic INtegrator) was written by Joel Moses in 1967, also as a doctoral dissertation at MIT. Whereas Slagle had compared the behavior of SAINT to that of freshman calculus students, Moses aimed at behaviour comparable to expert performance. It may be noted here that both of the above programs were coded in LISP.

1.1 AI LANGUAGES

AI programming languages have had a central role in the history of Artificial Intelligence, serving two important fuctions. First, they allow convenient implementation and modification of programs that demonstrate and test AI ideas. Second, they provide vehicles of thought: As with other highlevel languages,

they allow the user to concentrate on higher level concepts. Frequently, new ideas in AI are accompanied by a new language in which it is natural to apply these ideas. Some AI programming languages are IPL, LISP, PLANNER, CONNIVER, QLISP, POP-2, SAIL, FUZZY, PROLOG, etc.

1.1.1 LISP & PROLOG

The most established AI language is LISP, invented at MIT by John McCarthy in the 1950s. LISP is more convenient for AI work than conventional data-oriented languages. One reason is that it allows the direct representation of symbolic concepts and the relationships between them in the form of data structures called lists - infact lists are the only data structures in LISP. Another convenience of LISP is that it does not require the data types of each variable and the allocation of memory to each type to be specified at the beginning of the program; instead data types are determined at run time, and memecry is allocated flexibly according to requirements.

Besides its use of list structures as its primitive(and only) data types, LISP probably differs from other programming languages in its style of describing computations. Instead of fuctions defined in a rather mathematical format. Each function call is

represented as a list, the value of whose first element is the name of the function and the values of whose other elements are the arguments. Even though LISP is more suitable for AI work than conventional data-oriented languages, it has some drawbacks. They are

Ugly syntax: A common complaint about the list-structure format of LISP programs is that it makes them difficult to read. The only syntactic items are separators, such as spaces and parentheses, which provide most of the structure. This way of representing structures is convenient for machine to read, but inconvenient for humans.

Lack of language standard: Unlike FORTRAN and other well Known programming languages, there has never been an attempt to agree on a standardized LISP. The absence of a language standard and the proliferation of incompatible versions make LISP badly suited to be a production language, and in AI research work there are severe difficulties in transporting LISP programs to machines running a different LISP.

The other alternative to LISP suited to AI and symbolic computing developed by Alan Colmeraur in Europe in the 1970s is PROLOG. Prolog originated as an attempt to design a language which would allow the programmer to specify the objectives of a task interms of symbolic

logic. A major advantage of Prolog is that the expert systems concept of an inference engine working against a knowledge base, and seeking to satisfy assigned goals by fixing rules, maps very directly onto the language; in a sense any Prolog program can be seen as a sort of expert system.

A LISP program of a series of commands that manipulate symbols while a PROLOG program consists of statements of facts and rules. The powerful pattern matching capability and an automatic backtracking facility in PROLOG are an added advantage over LISP. PROLOG procedures are also flexible in the sense that the input and output parameters are not predetermined but may vary from call to call.

1.1.2 PROLOG Vs CONVENTIONAL LANGUAGES

PROLOG differs from the conventional languages in many aspects. A PROLOG program is predominantly "DECLARATIVE" in that it is concerned with stating WHAT has to be done, in the form of rules(logic) and facts, while a conventional program is more "PROCEDURAL" and concerned with HOW the task should be done. The conventional languages have similar data and program structures such as arrays, if_then_else and loops. There are no such constructs in Prolog. In conventional languages the programmer must specify step by step how a

result is to be computed. In constrast, in Prolog we describe what the relationships are among the entities. Prolog extensively uses recursion and a unique backtracking mechanism. Prolog variables do not represent storage locations. This means that all values assigned to variables are temporary for instantiation purposes and kept only for the duration of a specific execution of the clause. The programmer cannot increment a variable value as for example, N = N+1 is done in conventional languages. A Prolog procedure is a collection of rules rather than a single closed module of a subroutine.

2. INTRODUCTION TO THE PROJECT

Symbolic CAlculus Program (in subsequent sections, it is referred as SCAP, in brief) is a rule based program which identifies the input expression and evaluates the integral/derivative of the given expression. SCAP can be subdivided into two modules, integration & differentiation module and simplifiction module.

For a given input expression SCAP responds to it by performing following functions:

1. It invokes the integration/differentiation module.
2. It classifies the given input expression to one of the types and evaluates the integral/ differential of the input expression.
3. The integrated/differentiated expression is simplified, if neccessary.

Integration & differentiation module consists of two subtasks, namely integration and differentiation. The integration problems that SCAP could handle have only elementary fuctions as integrals. The domain of symbolic integration consists of following four types of problems:

1. Standard integrals - there are about 25 standard integrals. A typcial one indicated

that if the integrand has the form a^x dx, the form of the solution is ln(a)^(-1)*a^x.

2. Constant and a function - integral of a constant and a fuction is constant and an integral of a function. This function can recursively be of any of the four types again. Typically, integral of (a*cos(x))dx is a*sin(x), where 'a' is a constant with respect to x.

3. Integral of sum or difference of functions - that is, decomposing integral of sum/difference of functions into sum/difference of integrals. Here again each function can recursively be of any of the four types. Typically, integal of (x+e^x)dx is 2^(-1)*x^2+e^x.

4. Product of integrals or integration by parts - that is, given product of fuctions (other than '2') which can be integrable, its solution is evaluated by integration by parts, i.e., integral of U*V dx is U*integral(Vdx)-integral(differential(U)*integral(Vdx)dx). It may be noted here that differential of the first function has to be evaluated for integration by parts. This means to say that whenever the problem of integration by parts is encountered, differential routine is invoked and the respective function is differentiated. Typical

example is integral of (x*cos(x))dx is x*sin(x)+cos(x).

It may be noted here that most of the design effort has been spent on integration by parts and the description of the program is in next chapter. The whole symbolic integration problem i.e., above four types can be visualized as a tree shown in fig 2.1.

The program starts with the original problem as a goal, specified as an integrand and a variable of integration. For any particular goal, the strategy is to classify it as any one of the four types of integration, if it is in standard form then the solution is immediate, if it is not, it can be constant and a function, where function is a new goal to which the same strategy is applied, if it is not constant and a function, then it can be sum/difference of integrals, where two arguments of operators '+'/'-' are again treated as two new goals and the same strategy is applied, if it is not sum/difference of integrals, it can be product of integrals and solution obtained and if it is neither of the above four types the program simply cannot integrate.

The differentiation routine of SCAP gives the derivative of the given expression. The problem of differentiation is much simpler when compared to

integration as differentiation is much more systematic in nature than integration. Due to the systematic nature of the problem, there is only one type of rule in differentiation namely,

d(Exp, X, Result):

d - gives the derivative of expression (Exp) with respect to X as Result.

Exp - expression whose derivative is to be evaluated.

X - variable of differentiation.

Result - derivative of the given expn.

Listing of the above type of rule can be found in page p.17. Here unlike integration the control for the selection of the type of differentiation is included in the differentiation routine itself, meaning to say that there is no separate search strategy to classify to particular type of differentiation.

Simplification module in SCAP consists of different routines, each applicable for a particular type of simplification, which is encountered in integration/differentiation. While running the SCAP different simplification routines are invoked at different levels of integration/differentiation according to their need in the execution.

Two major rule types used in simplification are as follows:

RULE I

symty1(Expn, Sexpn):

symty1 - this is a predicate name for different types of simplifications having two arguments Expn & Sexpn. Simplification routines of this type are simp ssimp & sp.

Expn - expression to be simplified.

Sexpn - simplified expression.

Listing of these rules are given in pages p.1-2, p.6-8.

RULE II

symty2(Expn, Var, Sexpn):

symty2 - this is a predicate name for different types of simplifications having three arguments, namely Expn, Var & Sexpn. Simplification routines of this type are simpl, trig_simp.

Expn - expression to be simplified.

Var - variable of integration/ differentiation used in simplification routines to determine constants in the given expression.

Sexpn - simplified expression.

Listing of these rules are given in pages p.3-6, and explanation is in next chapter.

Standard integrals – c_1

Constant and a function – c_2 : $a \times f(x)$

sum/difference of integrals – c_3 : $f(x) \pm g(x)$

product of integrals – c_4 : $u \times v$

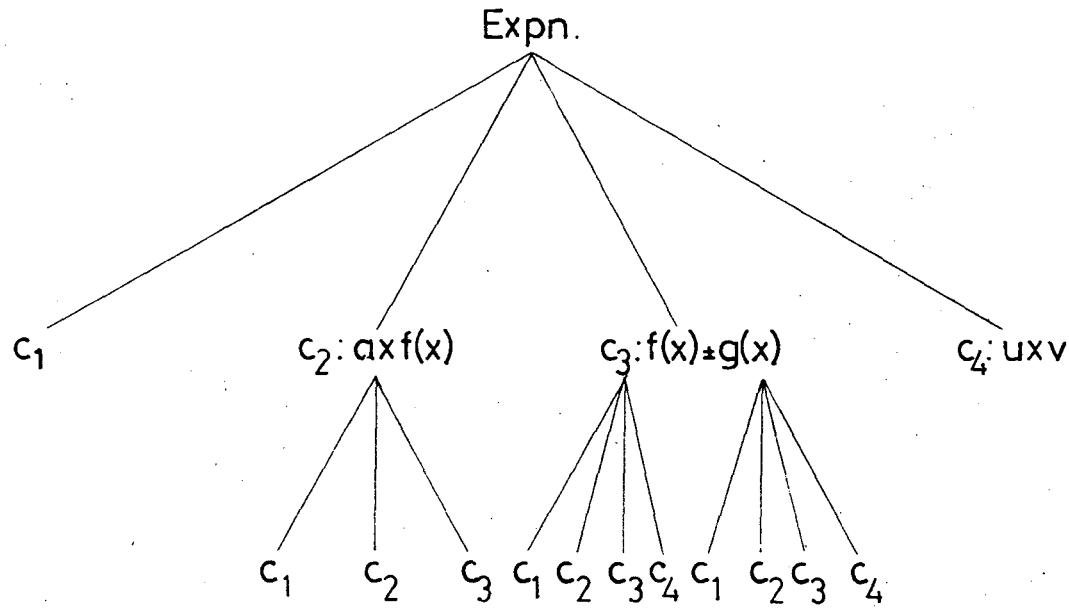

fig 2.1

3. DESCRIPTION OF THE PROGRAM

3.1 INTEGRATION & DIFFERENTIATION

As explained in the previous chapter, given an expression how the integration module will classify it into one of the four types of integration. This classification to particular type of integration is done by check_int(Expn, X, Result) routine. This routine checks the expression(Expn) for the type of integration and integrates the given expression with variable of integration as X to give integral as Result. Refer to page p.9 for listing of this routine.

Rules for the four types of integration are represented as shown below:

typ_int(Expn, Var, Result):

typ_int - denotes predicate name for type of integration, namely,

stand_int - standard integral

const_and_int - constant and an integral

sum_of_ints / diff_of_ints - sum/difference of integrals

int_by_parts - integration by parts

Expn - given integrand

Var - variable of integration

Result - integral of Expn

The above four types of integration rules are explained in detail below, each separately.

Standard integrals:

There are about 25 standard integrals each represented as follows

int(Stdingd,Var,Int):

int - predicate name to indicate integral of

Stdingd - standard integrand whose integral is immediately known

Var - variable of integration

Int - integral of standard integrand

For example

int(x^3,x,4^(-1)*x^4).

Listing of this type of rules are given in pages p.9-10.

Constant and an integral:

In this routine i.e. const_and_int, integral of the given expression is determined only if the given expression is product of two terms and the left handside term (or the first term) is a constant with respect to variable of integration and right hand term (or the second term) is again matched to either one of the three types, namely, stand_int, const_and_int, sum_of_ints or diff_of_ints. The boundary condition for

this routine to succeed is that the second term is a standard integrand. It is to be noted here that user is always requested to give the input expression in the cannonical form

i.e. [numbers]operator[constants]operator[f(x)]

so that the order of the terms in the expression is maintained throughout integration/differentiation. Refer to page p.11 for listing of this routine.

Typical example,

const_and_int(2*x^(-1),x,2*ln(x))

Sum/difference of integrals:

In this routine i.e. sum_of_ints/diff_of_ints, given expression is examined if it is sum/difference of two terms. These two terms are treated as two subgoals which can again recursively be of any one of the four types of integration.

For example,

diff_of_ints(cos(2*x)-e^x,x,2^(-1)*sin(2*x)-e^x).

sum_of_ints(sec(x)^2+2*x^2,x,tan(x)+2*3^(-1)*x^3).

Refer to page p.11 for listing of this routine.

Integration by parts:

In this routine i.e. int_by_parts, if the given expression is product of two terms, i.e. other than a constant and a function, then it is treated as a problem

of integration by parts and accordingly its integral is evaluated.

For an expression U*V to be integrated by the method of integration by parts, first and foremost criteria is to select the order UV, such that

(i) integral of V be simple
(ii) integral of du/dx be simple

The routine, select_the_order(U*V,X,R) selects the order of the expression U*V and instantiates the ordered expression to variable R. The method for ordering the expression is as follows

```
If      { U = sin(Y) or cos(Y) or tan(Y) or cot(Y) or
            sec(Y) or cosec(Y) } OR
        { U = e^Y or a^Y(where 'a' is a constant w.r.to
            x) and V = X^Z } OR
        { U = sec(Y)^2 or cosec(Y)^2 } OR
        { V = ln(Y) } OR
        { V = ln(Y)^N } OR
        { V = e^Y and U \= X^Z }
then
        select_the_order(U*V,X,V*U)
else
        select_the_order(U*V,X,U*V).
```

Listing of this routine can be found in page p.12.

Now the ordered expression is to be integrated and this is done by eval_int(Expn,X,Result) routine and the value of the Result is passed on to simplification routine ssimp(E,R) (this routine is explained in simplification module in next section) and the result 'R' is the integral of the given expression.

The eval_int(U*V,X,Result) procedure approach the problem of integration in two different ways depending upon two conditions, they are

(i) V is a standard integrand

(ii) V is not a standard integrand

i.e. here we consider certain problems like

e^x*(tan(x)-ln(cos(x)),

e^x*((1+sin(x))/(1+cos(x)),

e^x*((x-1)/(x+1)^3),

etc.,

where 'V' is not a standard integrand.

First consider the second case, here problems are of the type

integral(e^x*(f(x)+f'(x)) = e^x*f(x).

The method for this type of problems is follows

(i) Check if the first term(U) in the expression(U*V) is e^x

(ii) if yes, check if the second term(V) is directly of the form "f(x)+f'(x)", if yes, f(x) is passed on

and the integral of the given expression is e^x*f(x).

The routine test_arg1_arg2(V,X,R) will test if the second argument (f'(x)) of V is the derivative of the first argument (f(x)) or vice versa and if it succeeds then R is instantiated to f(x).

Typically

test_arg1_arg2(sin(x)+cos(x),x,sin(x)).

Refer to page p.16 for listing of this routine.

(iii) if V is not directly of the form "f(x)+f'(x)", then if possible, it is transformed to this particular form and its integral is evaluated.

Typically,

```
(1+sin(x))/(1+cos(x))
                --)   (1+2*sin(x/2).cos(x/2))/2.cos(x/2)^2
                --)    1/2.sec(x/2)^2+tan(x/2)
(x-1)/(x+1)^3   --)    (x+1-2)/(x+1)^3
                --)    (x+1)^-2 - 2.(x+1)^-3
```

For example

eval_int(e^x*((1+sin(x))/(1+cos(x)),x,e^x*tan(2^(-1)*x))

The above procedure is coded in the last three clauses of the eval_int routine and listed in pages p.13-14. In this routine trig_simp(Expn,X,Sexpn) procedure is used to simplify the trignometric functions Expn (encountered

in the above type of problems) to Sexpn (This procedure is explained in detail in simplification module in next section).

Now consider the first case where in 'V' is a standard integrand.

We know that

```
Int(U*V)dx = U*Int(Vdx)- Int(U'*Int(Vdx)dx)
             |________|  |________________|
                  ^             ^
                  E1            E2

           = U*V1 - Int(U1*V1dx)
             |__|   |__________|
              ^          ^
              R1         R2
```

In this case E1 is easily evaluated to R1, R1 is stored in a list, say Oldlist and E2 is evaluated to R2. As one can notice, R2 is again a problem of integration by parts, which is recursive, so a new routine test2_int is used which will append all R1's of R2 with Oldlist to form Newlist.

The format of the rule is as follows

test2_int(NE,X,OE,OL,NL):

NE - new expression which is to be integrated i.e. R2

X - variable of integration

OE - original expression i.e. U*V

OL - oldlist i.e. [R1] here

NL - new list formed by appending

oldlist with R1's obtained from NE

The boundary conditions for this recursive routine are

1. NE is either a constant and a function or a standard integrand
2. NE is equivalent to OE or

   if NE = Constant * NE1 then NE1 is equivalent to OE here [Constant] is appended with R1's to form Newlist This is particularly useful for integrands like, e^x*cos(x), a^x*sin(x), etc.

   If NE = -NE1, then [-] is appended with Oldlist to form Newlist. Refer to page p.14 for listing of this routine.

For example

```
test2_int(3*x^2*sin(x),x,x^3*cos(x),[x^3*sin(x)],
        [x^3*sin(x),-3*x^2*cos(x),-,6*x*sin(x),
        -6*cos(x)])
test2_int(e^x*sin(x),x,e^x*cos(x),[e^x*sin(x)],
        [e^x*sin(x),-e^x*cos(x),-,1])
```

Consider

Int(U*Vdx) = U*Int(Vdx) - Int(U'*Int(Vdx)dx)

It is to be noted here that the negative operator '-' is not an element in the Newlist, so when evaluating the list this negation must be considered. The order of the elements in the above list is reversed for evaluating the list. The eval_list(List,X,R) evaluates the reversed

list(List) to give the result as R. The head and tail list of the list(List) is obtained and the tail list is checked for four conditions given below by check_tail_eval routine and depending on the conditions satisfied , integral is evaluated which is passed onto eval_list routine.

Rule type

check_tail_eval(T, X, H, R):

check_tail_eval - checks the tail list and accordingly evaluates the list to give result as R

T - tail list of the original list

X - variable of integration used here to determine constants

H - old result of the list

R - final result of the original list

Refer to pages p.15-16 for listing of this routine.

Let T be the tail list and H be the head of the original list again T1 and H1 be the tail and the head of the list T.

(i) if H is not a constant, H1 \== '-', T1 is an empty list then evaluated result of the original list is a boundary condition for check_tail_eval routine.

(ii) if H is a constant and

(a) if T1 is an empty list , then R = H1, OR

(b) if H1 is equivalent to '-', then eval_list(T1,X,R), OR

(c) if H1 is not equivalent to '-', then eval_list(T,X,R) and simplify (1+H)^(-1) and instantiate it to a variable(Const), then final result of list is Const*R.

(iii) if H1 is not equivalent to '-', T1 is not an empty list, then R = simplified H1 - H and again evaluate the remaining tail list T1 until it is empty by check_tail_eval(T1,X,R,Result) routine to give final result as Result.

(iv) if H1 is equivalent to '-', then R is equal to -(H) and again evaluate the ramaining tail list T1 until it is empty by check_tail_eval(T1,X,R,Result) to give final result as Result.

For example

```
eval_list([-6*cos(x),6*x*sin(x),-,-3*x^2*cos(x),
      x^3*sin(x)],x,x^3*sin(x)-(-3*x^2*cos(x)-
      (6*x*sin(x)-(-6*cos(x))))).
eval_list([1,-,-e^x*cos(x),e^x*sin(x),x,
      2^(-1)*(e^x*sin(x)-(-e^x*cos(x)))).
```

The result obtained above is simplified by ssimp routine and gives the integral of the given expression,

i.e.

```
int_by_parts(x^3*cos(x),x,x^3*sin(x)+3*x^2*cos(x)
```

```
-6*x*sin(x)-6*cos(x)).
```

```
int_by_parts(e^x*cos(x),x,2^(-1)*(e^x*sin(x)+e^x*cos(x))
```

Differentiation routine of SCAP gives the derivative of the given expression and the type of the rule for this routine is already explained in the previous chapter.

Typically

```
d(ln(x),x,x^(-1)) - derivative of ln(x) is x^(-1)
```

with respect to x.

3.2 SIMPLIFICATION

In SCAP, simplification module contains different routines each applicable for a paraticular type of simplifiction. Before proceeding to describe these routines, let us consider a procedure const(Y,X) which determines if 'Y' is a constant with respect to 'X'. 'Y' is a constant with respect to 'X' if 'Y' does not contain 'X'. Refer to page p.1 for listing of this procedure.

The routine simp(Expn,Sexpn) simplifies the expression Expn to simplified form Sexpn. In this routine, separate rules (& facts) are written for elementary simplifications, like

(i) 0*0 = 0

(ii) X*1 = X, X*1 = X, X*X = X^2, etc.

(iii) X+0 = X, X+X = 2*X, etc.

(iv) X-0 = X, 0-X = -X, X-X = 0, etc.

(v) (X^M)^N = X^(M*N)

and so on.

In this routine, given expression is examined if it matches into any one of the elementary simplifications(like those listed above) and if it does, then the expression is immediately simplified. Otherwise given expression is split into main operator(according to precedence of operators) & arguments and recursively for each argument the simp routine is applied and the simplified expression of the original expression is obtained.

For example

simp(x*x^2 + x^0, x^3+1)

Listing of this type of routine is found in pages p.1-2.

The routine, trig_simp transforms trignometric functions such as sin(Y)^(-N), cos(Y)^(-N), etc. to cosec(Y)^N, sec(Y)^N, etc. and vice versa. This type of trignometric transformations are used in some problems involving integration by parts in SCAP.

For example

trig_simp(sin(x)^(-1), x, cosec(x)).

Refer to page p.3 for listing of this routine.

The routine, simpl(E,X,SimpE) simplifies product of expressions E to SimpE, which is in the cannonical form,

i.e. [numbers]*[constants]*[f(x)]

In this routine given expression is split into three lists, namely numberlist, constantlist(excluding numbers) and non constantlist. Each element in these lists is again a list consisting of two elements, first element is the base and the second element is the exponent. Once the three lists are formed, each list is evaluated individually and the final simplified form of the given expression will be

value of numberlist * evaluated constantlist * evaluated non constantlist

To evaluate the number list, each element's(i.e. list consisting of two elements) value is evaluated and the product of all the elements values in the number list will be the value of the number list.

For example

consider a number list be of the form

[[2,1],[3,2],[9,1]]

value of each element is 2^1 = 2

3^2 = 9

9^1 = 9

and the value of the number list is

2*9*9 = 162

Procedure to evaluate constantlist and non constantlist:

First element of the list is taken. This element is a list consisting of two elements b1-base and e1-exponent i.e. [b1,e1]. Now all the elements having b1 as the base are taken and simplified to b1^(e1+e2+e3+...) and the remaining elements are stored in a newlist. Same procedure is applied for newlist and it will continue until the newlist is empty.

For example

consider constant list consisting of following elements

[[3,a],[a,2],[3,b],[a,4]]

First consider the first element of the list i.e. '3', take all elements having base as '3', they are

[3,a] and [3,b]

and its value is 3^(a+b)

Now the newlist is [[a,2].[a,4]]

base of the firtst element is 'a'.

consider all elements having base as 'a', they are

[a,2] and [a,4]

and its value is a^(2+4) = a^6

now the newlist is empty

so the simplified form of the evaluated constant list is 3^(a+b)*a^6. Rules corresponding to the above method for

simpl routine are listed in pages p.3-6.

Example,

Let the given expression be

2*a^2*x^2*sin(x)*4^2*ln(a)*x^(-1)

three lists formed are as follows

Number list - [[2,1],[4,2]]

Constant list - [[a,2],[ln(a),1]]

Non constant list - [[x,2],[sin(x),1],[x,-1]]

Numberlist:

value of each element is 2^1 = 2

4^2 = 16

value of number list is 2*16 = 32

Constant list:

all elements having base as 'a' = [a,2]

its value is a^2

new list is [[ln(a),1]]

all elements having base as ln(a) = [ln(a),1]

its value is ln(a)^1 = ln(a)

new list is empty

Simplified form of the constant list is a^2*ln(a)

Non constant list:

all elements having base as 'x' are [x,2] and [x,-1]

its value is x^(2+(-1)) = x^1 = x

new list is [[sin(x),1]]

all elements having base as 'sin(x)' = [sin(x),1]

its value is sin(x)^1 = sin(x)

```
 newlist is empty
 Simplified form of the Nonconstant list is x*sin(x)
Simplified form of the given expression is
      32*a^2*ln(a)*x*sin(x)
```

The routine ssimp(Expn, Sexpn) simplifies the signed expression Expn to Sexpn, with special reference to sign simplification encountered in integration by parts. First the given expression is simplified for unary negation and for the principal negative operator. This negative sign simplification is done by nssimp routine. Output of this routine is an expression consisting of principal operator as plus. Now the pssimp routine is invoked which simplifies the outputed expression from nssimp routine to give the final simplified expression.

For example

```
ssimp(x^3*sin(x)-(-3*x^2*cos(x)-(-6*x*sin(x)
 -(-(6*cos(x)))),x^3*sin(x)+3*x^2*cos(x)-6*x*sin(x)
 +6*cos(x)).
```

4. LIKELY IMPROVEMENTS

SCAP has the flexibility to incorporate other methods of integration not considered here, like, integration by substitution, integration by partial fractions, etc. For this, SCAP has to be enhanced with following routines. They are

(i) New simplification routines:

New simplification routines are to be incorporated in the simplification module which will simplify & transform the given expression according to the type of integration to which it has been classified by classification routine. Typically, if the given expression has been classified as a problem of integration by partial fractions, simplification routines are to be invoked which will transform the expression to partial fractions for direct integration.

(ii) Classification routine:

This routine will identify and classify the given expression to one of the methods of integration.

Consider cos(x)^2

integral of cos(x)^2 cannot be evaluated directly, so the new simplification routine should transform or substitute cos(x)^2 to

cos(2*x)/2+1/2

now this transformed expression can directly be integrated to give the integral of cos(x)^2.

5. LISTING OF THE PROGRAM

```
/*-------------------------------------------------------------*/

const(Y,X):- atomic(X),atomic(Y),(X\==Y,!;!,fail).
const(X,X):-!,fail.
const(E,X):- E=..[_,Z],!,const(Z,X).
const(E,X):- arg(1,E,Left),!,const(Left,X),arg(2,E,Right),!,
             const(Right,X).
const(_,_).

/*-------------------------------------------------------------*/

/*** simp(E,SimpE): simplifies expression(E) to SimpE ***/

simp(E,E):- atomic(E),!.
simp(E,SimpE):- E=..[O,Z],simp(Z,Z1),SimpE=..[O,Z1],!.
simp(E,SimpE):- E=..[O,L,R],number(L),number(R),
                (
                 (O=='^',name(R,[45|T1]),T1\==[],L\==1,L\==0,
                  (
                   (
                    name(L,[45|T2]),name(P,T1),name(Q,T2),
                                                Z is P/2,
                   (integer(Z),SimpE = Q^R;SimpE = -Q^R)
                   );
                   SimpE = E
                  )
                 );
                 SimpE is E
                ),!.

simp(M*X+N*X,R):-
                  (
                   (number(M),number(N), K is M+N);
                   (number(M),simp(M+N,K));
                   (number(N),simp(N+M,K));
                   simp(M+N,K)
                  ),
                  simp(K*X,R),!.

simp(M*X-N*X,R):-
                  (
                   (number(M),number(N),K is M-N);
                   (number(M),simp(M-N,K));
                   (number(N),simp(N-M,K));
                   simp(M-N,K)
                  ),
                  simp(K*X,R),!.
```

```
simp(X+Y,SR):- (Y=0,R=X;X=0,R=Y;X=Y,R=2*X),simp(R,SR),!.
simp(X-Y,SR):- (((Y=0,R=X;X=0,R= -Y),simp(R,SR));X=Y,SR=0),!.
simp(X*Y,0):- (X=0;Y=0),!.
simp(X*Y,SR):- (Y=1,R=X;X=1,R=Y;X=Y,R=X^2),simp(R,SR),!.
simp(X/X,1).
simp((-X)*(-Y),R):- simp(X*Y,R),!.
simp((-X)*Y,R):- simp(X*Y,R1),R = -R1,!.
simp(X*(-Y),R):- simp(X*Y,R1),R = -R1,!.
simp(X^0,1):-!.
simp(X^1,X):-!.
simp(X^N*X,R):-
                  (
                   (number(N),K is N+1);
                    simp(1+N,K)
                  ),
                  simp(X^K,R),!.

simp(X*X^N,R):- simp(X^N*X,R),!.
simp(X^M*X^N,R):-
                   (
                    (number(M),number(N),K is N+M);
                    (number(M),simp(M+N,K));
                    (number(N),simp(N+M,K));
                     simp(M+N,K)
                   ),
                   simp(X^K,R),!.

simp((X^M)^N,R):-
                   (
                    (number(M),number(N),K is M*N);
                    (number(M),simp(M*N,K));
                    (number(N),simp(N*M,K));
                     simp(M*N,K)
                   ),
                   simp(X^K,R),!.

simp((X*Y)^N,R):-
                   (
                    (X==Y,simp((2*X)^N,R));
                    (simp(X*Y,X*Y),R = X^N*Y^N);
                    (simp(X*Y,P),simp(P^N,R))
                   ),!.
simp(E,SimpE):- E=..[O,L,R], atomic(L),atomic(R),SimpE=E,!.
simp(E,SimpE):- E=..[O,L,R],simp(L,SL),simp(R,SR),
                  TimpE=..[O,SL,SR],
                  (
                   (L\==SL;R\==SR),
                   simp(TimpE,SimpE);SimpE=TimpE
                  ),!.

simp(E,E).
```

```
/*-------------------------------------------------------------*/

/*** trig_simp(E,X,SimpE): simplifies trignometric expression
     (E) to SimpE ***/

trig_simp(E,X,SimpE):-

                E=..[*,L,R],const(L,X),
                trig_simp(R,X,SimpR1),
                SimpE=..[*,L,SimpR1].

trig_simp(E,X,SimpE):-

                E=..[^,L,R1],number(R1),
                L=..[F,A],
                (
                 (F==sin,R2=..[cosec,A]);
                 (F==cos,R2=..[sec,A]);
                 (F==tan,R2=..[cot,A]);
                 (F==cot,R2=..[tan,A]);
                 (F==sec,R2=..[cos,A]);
                 (F==cosec,R2=..[sin,A])
                ),
                N is -(R1),
                SimpE=..[^,R2,N].


/*-------------------------------------------------------------*/

/*** simpl(E,X,SimpE): simplifies product of expressions(E)
     to SimpE which is in the cannonical form
     i.e., [numbers]*[constants]*[f(x)] ***/

simpl(E,X,R):- simp(E,SimpE),simpl1(SimpE,X,R).

simpl1(-E,X,-R):- simpl1(E,X,R),!.

simpl1(E,X,R):-

                E =..[*,E1,E2],
                formlists(E1,X,[],Nlist1,[],Clist1,[],
                                                NClist1),
                formlists(E2,X,[],Nlist2,[],Clist2,[],
                                                NClist2),
                append(Nlist1,Nlist2,Nlist),
                append(Clist1,Clist2,Clist),
                append(NClist1,NClist2,NClist),
                eval_Nlist(Nlist,N),
                eval_Olist(Clist,C),
                eval_Olist(NClist,NC),
                Y = N*C,simp(Y,SimpY),
                Z = SimpY*NC,
```

```
                    simp(Z,R).

simpl1(E,X,E):- !.

/*** formlists(E,X,O_NL,N_NL,O_CL,N_CL,O_NCL,N_NCL):
     splits the expression E into three lists:
     Numberlist(NL),Constantlist(CL), Nonconstantlist(NCL).
     O & N represents Old & New lists ***/

formlists(P*Q,X,Old_Nlist,New_Nlist,Old_Clist,New_Clist,
                        Old_NClist,New_NClist):-

                    formlists(P,X,Old_Nlist,New_NlistP,
                         Old_Clist,New_ClistP,Old_NClist,
                                          New_NClistP),
                    formlists(Q,X,[],New_NlistQ,[],New_ClistQ,
                                    [],New_NClistQ),
                    append(New_NlistP,New_NlistQ,New_Nlist),
                    append(New_ClistP,New_ClistQ,New_Clist),
                    append(New_NClistP,New_NClistQ,New_NClist),
                    !.

formlists(E,X,Old_Nlist,New_Nlist,Old_Clist,New_Clist,
                        Old_NClist,New_NClist):-

                    (
                     (number(E),
                      append(Old_Nlist,[E],New_Nlist));
                     (E =..[^,P,Q],
                      number(P),number(Q),
                      (P==1;P==0;name(Q,[_|T]),T==[]),R is E,
                      append(Old_Nlist,[R],New_Nlist))
                    ),
                    append(Old_Clist,[[1,1]],New_Clist),
                    append(Old_NClist,[[1,1]],New_NClist),!.

formlists(E,X,Old_Nlist,New_Nlist,Old_Clist,New_Clist,
                        Old_NClist,New_NClist):-

                    (
                     (E =..[^,P,Q],
                      const(P,X),const(Q,X),
                      append(Old_Clist,[[P,Q]],New_Clist));
                     (const(E,X),append(Old_Clist,[[E,1]],
                                          New_Clist))
                    ),
                    append(Old_Nlist,[1],New_Nlist),
                    append(Old_NClist,[[1,1]],New_NClist),!.

formlists(E,X,Old_Nlist,New_Nlist,Old_Clist,New_Clist,
                        Old_NClist,New_NClist):-
```

```
                    not const(E,X),
                    (
                     (E =..[^,P,Q],
                      append(Old_NClist,[[P,Q]],New_NClist));
                     append(Old_NClist,[[E,1]],New_NClist)
                    ),
                    append(Old_Nlist,[1],New_Nlist),
                    append(Old_Clist,[[1,1]],New_Clist),!.

/*** eval_Nlist(L,R): evaluates the number list L to
     give result as R ***/

eval_Nlist([],1):- !.
eval_Nlist([H|T],Result):- testtail_evalNlist(T,H,Result).

/*** testtail_evalNlist(L,IR,FR): tests the tail of the
     list(L) and evaluates to give final result FR from
     the initial result IR ***/

testtail_evalNlist([],H,H):- !.

testtail_evalNlist([H1|T1],H,Result):-

                    NewH is H*H1,
testtail_evalNlist(T1,NewH,Result).

/*** eval_Olist(L,R): evaluates the other lists
     i.e.,Constant & Nonconst lists to give result as R ***/

eval_Olist([],1):- !.
eval_Olist(List,Result):- evaleachE_withnextE(List,1,Result).

evaleachE_withnextE([],R,R):- !.

evaleachE_withnextE(List,IResult,FResult):-

                    first_two_Es_of(List,RList,E1,E2),
                    equal_base_test(E1,E2,RList,[],NewList,
                                                    Result),
                    simp(IResult*Result,Result1),
                    evaleachE_withnextE(NewList,Result1,
                                                    FResult).

first_two_Es_of(List,NList,P,Q):-

                    first_E_of(List,NList1,P),
                    first_E_of(NList1,NList,Q).

first_E_of([],[],[]):- !.
first_E_of([H|T],T,H):- !.

equal_base_test([Bs1,Exp1],[],[],Oldlist,Oldlist,R):-
```

```
                    simp(Bs1^Exp1,R),!.

equal_base_test([Bs1,Exp1],[Bs2,Exp2],RList,Oldlist,
                                        Newlist,Result):-

                    atomic(Exp1),atomic(Exp2),
                    name(Exp1,[45|T1]),T1\==[],
                    name(Exp2,[45|T2]),T2\==[],
                    (
                     (number(Exp1),number(Exp2),
                           Exp is Exp1+Exp2);
                     simp(Exp1+Exp2,Exp)
                    ),
                    number(Bs1),number(Bs2),Bs is Bs1*Bs2,
                    first_E_of(RList,NewRList,E3),
                    equal_base_test([Bs,Exp],E3,NewRList,
                                    Oldlist,Newlist,Result).

equal_base_test([Bs1,Exp1],[Bs1,Exp2],RList,Oldlist,
                                        Newlist,Result):-

                    simp(Exp1+Exp2,Exp),
                    first_E_of(RList,NewRList,E3),
                    equal_base_test([Bs1,Exp],E3,NewRList,
                                    Oldlist,Newlist,Result).

equal_base_test(E1,E2,RList,Oldlist,Newlist,Result):-

                    append(Oldlist,[E2],Oldlist1),
                    first_E_of(RList,NewRList,E3),
                    equal_base_test(E1,E3,NewRList,Oldlist1,
                                        Newlist,Result).

append([],L,L).
append([X|L1],L2,[X|L3]):- append(L1,L2,L3).


/*-------------------------------------------------------------*/


sp(E,R):-

          functor(E,-,_),
          arg(1,E,E1),arg(2,E,E2),
          check_num(E1,Number1,Atom1),
          check_num(E2,Number2,Atom2),
          Number is Number1-Number2,
          Atom = Atom1-Atom2,simp(Atom,SAtom),
          SimpE = Number+SAtom,simp(SimpE,R).

sp(E,E):-!.
```

```
check_num(E,E,0):- number(E),!.

check_num(E,0,E):- atomic(E),!.

check_num(E,Number,SAtom):-

        functor(E,+,_),
        arg(1,E,P),arg(2,E,Q),
        check_num(P,NumberP,AtomP),
        check_num(Q,NumberQ,AtomQ),
        Number is NumberP+NumberQ,
        Atom = AtomP+AtomQ,simp(Atom,SAtom).

check_num(E,Number,SAtom):-

        functor(E,-,_),
        arg(1,E,P),arg(2,E,Q),
        check_num(P,NumberP,AtomP),
        check_num(Q,NumberQ,AtomQ),
        Number is NumberP-NumberQ,
        Atom = AtomP-AtomQ,simp(Atom,SAtom).

/*-------------------------------------------------------------*/

/*** ssimp(E,SE): simplifies the signed expression
                  E to SE ***/

ssimp(P*Q,SP*SQ):- ssimp(P,SP),ssimp(Q,SQ).
ssimp(Exp,SExp):- nssimp(Exp,X),pssimp(X,SExp).

nssimp(-(-U+V),U1+R):- nssimp1(U,U1),nssimp1(-V,V1),
                                        nssimp(V1,R).
nssimp(-(U+V),U1+R):- nssimp1(-U,U1),nssimp1(-V,V1),
                                        nssimp(V1,R).
nssimp(-(-U-V),U1+R):- nssimp1(U,U1),nssimp1(V,V1),
                                        nssimp(V1,R).
nssimp(-(U-V),U1+R):- nssimp1(-U,U1),nssimp1(V,V1),
                                        nssimp(V1,R).
nssimp(U-V,U1+R):- nssimp1(U,U1),nssimp1(-V,V1),nssimp(V1,R).
nssimp(U,U):- !.

nssimp1(-(-U),R):- nssimp1(U,R),!.
nssimp1(-(U),-U):- !.
nssimp1(U,U):- !.


pssimp(U+(-V+W),R):- pssimp(U-V+W,R).
pssimp(U+(V+W),R):- pssimp(U+V+W,R).
pssimp(U+(-V),U-V):- !.
pssimp(+(+U),U):- !.
pssimp(+(-U),-U):- !.
```

```
pssimp(+(U),U):- !.
pssimp(U,U):- !.
```

```
/*-------------------------------------------------------------*/

integrate :-    tell(t3),readfile(file),
                repeat,
                integ,nl,nl,
    print('Do you want to integrate another expression? '),
                read(Ans),
                nl,
                ((Ans\==yes,true);fail).

integ :-        nl,
    print('Give the expression to be integrated : '),
                read(E),Y =..[check_int,E,x,Result],
                call(Y),nl,nl,
                print( 'Integral of' ),tab(3),
                print('"'),print(E),print('"'),
                tab(2),print(' is = '),tab(2),
                print(Result),!.

/*-------------------------------------------------------------*/

/*** check_int(Exp,X,R): checks the expression(Exp) and
     integrate Exp w.r.to X to give R ***/

check_int(-E,X,R):- check_int(E,X,R1),R = -R1.

check_int(U/V,X,R):- simp(U*V^(-1),E),check_int(E,X,R).

check_int(E,X,R):-
                stand_int(E,X,R);
                const_and_int(E,X,R);
                sum_of_ints(E,X,R);
                diff_of_ints(E,X,R);
                int_by_parts(E,X,R).

/*-------------------------------------------------------------*/

/*** stand_int(Exp,X,R): expression(Exp) is a standard
     integrand integral of Exp w.r.to X is R ***/

stand_int(E,X,R):- int(E,X,R1),simpl(R1,X,R).

/*** int(Exp,X,R): gives integral of expression(Exp)
     w.r.to X as R ***/

int(A,X,A*X):- const(A,X),!.
int(X^(-1),X,ln(X)):- !.
int(X,X,P):- int(X^1,X,P),!.
int(e^X,X,e^X):- !.
int(e^U,X,P):- int(e^U,U,R),diff(U,X,U1),const(U1,X),!,
```

```
                P = U1^(-1)*R.
int(X^N,X,M^(-1)*X^M):- const(N,X),
                        (number(N),M is N+1,!;M = N+1).
int(A^X,X,ln(A)^(-1)*A^X):- const(A,X),!.
int(A^(U),X,P):- const(A,X),int(A^U,U,R),diff(U,X,U1),
                 const(U1,X),!,P = U1^(-1)*R.
int(-T,X,-P):- int(T,X,P).
int(sin(X),X,-cos(X)):- !.
int(cos(X),X,sin(X)):- !.
int(sec(X)^2,X,tan(X)):- !.
int(sec(U)^2,X,R):- int(sec(U)^2,U,R1),diff(U,X,U1),
                    const(U1,X),R = U1^(-1)*R1,!.
int(cosec(X)^2,X,-cot(X)):- !.
int(cosec(U)^2,X,R):- int(cosec(U)^2,U,R1),diff(U,X,U1),
                      const(U1,X),R = U1^(-1)*R1,!.
int(sec(X)*tan(X),X,-sec(X)):- !.
int(cosec(X)*cot(X),X,-cosec(X)):- !.
int(tan(X),X,ln(sec(X))).
int(tan(X),X,-ln(cos(X))):-!.
int(cot(X),X,ln(sin(X))).
int(cot(X),X,-ln(cosec(X))):-!.
int(sec(X),X,ln(sec(X)+tan(X))):-!.
int(cosec(X),X,ln(cosec(X)-cot(X))):-!.
int(E,X,R):- E =..[O,U],O\=='-',U\==X,!,int(E,U,R1),
             diff(U,X,U1),const(U1,X),!,R = U1^(-1)*R1.
int(U^(-1),X,P):- int(U^(-1),U,R),diff(U,X,U1),const(U1,X),!,
                  P = U1^(-1)*R,!.
int(U^N,X,P):- number(N),!,int(U^N,U,R),diff(U,X,U1),
               const(U1,X),!,P = U1^(-1)*R,!.
int((1-X^2)^(-(2^(-1))),X,arcsin(X)):- !.
int((A^2-X^2)^(-(2^(-1))),X,P):- integer(A),
                                 P = arcsin(X*(A)^(-1)).
int((1+X^2)^(-1),X,arctan(X)):- !.
int((X^2+1)^(-1),X,arctan(X)):- !.
int((X^2+A^2)^(-1),X,P):- integer(A),
                          P = arctan(X*A^(-1)*A^(-1)).
int((A^2+X^2)^(-1),X,P):- int((X^2+A^2)^(-1),X,P).
int((X^2+A)^(-(2^(-1))),X,ln(X+(X^2+A)^(2^(-1)))):-
                                                integer(A).
int((A+X^2)^(-(2^(-1))),X,P):- int((X^2+A)^(-(2^(-1))),X,P).
int((X^2-A)^(-(2^(-1))),X,ln(X+(X^2-A)^(2^(-1)))):-
                                                integer(A).
int((X^2-A^2)^(-1),X,P):- integer(A),S is 2*A,
                          P = S^(-1)*ln((X-A)*(X+A)^(-1)).
int((A^2-X^2)^(-1),X,P):- integer(A),S is 2*A,
                          P = S^(-1)*ln((A+X)*(A-X)^(-1)).

/*-------------------------------------------------------------*/

/*** const_and_int(Exp,X,R): expression(Exp) is of the form
     (constant)*(expression) ***/
```

```
const_and_int(E,X,Result):-

                E =..[*,P,E1],
                const(P,X),!,
                test_int(E1,X,R1),
                R = P*R1,
                simpl(R,X,Result).

test_int(E,X,R):-

                stand_int(E,X,R);
                const_and_int(E,X,R);
                sum_of_ints(E,X,R);
                diff_of_ints(E,X,R).

/*-------------------------------------------------------------*/

/*** sum_of_ints(Exp,X,R): expression(Exp) is of the form
     (exp1+exp2) i.e.,sum of integrals ***/

sum_of_ints(U+V,X,U1+V1):-

                test1_int(V,X,V1),
                test1_int(U,X,U1).

/*** diff_of_ints(Exp,X,R): expression(Exp) is of the form
     (exp1-exp2) i.e.,difference of integrals ***/

diff_of_ints(U-V,X,U1-V1):-

                test1_int(V,X,V1),
                test1_int(U,X,U1).

test1_int(U,X,U1):-

                U =..[/,P,Q],
                simpl(Q^(-1)*P,X,SimpU),
                test1_int(SimpU,X,U1),!.

test1_int(U,X,U1):-

                stand_int(U,X,U1);
                const_and_int(U,X,U1);
                sum_of_ints(U,X,U1);
                diff_of_ints(U,X,U1);
                int_by_parts(U,X,U1).

/*-------------------------------------------------------------*/

/*** int_by_parts(Exp,X,R): expression(Exp) is of the form
     p(x)*q(x) i.e.,integration by parts ***/
```

```
int_by_parts(U*V,X,Result):-

                select_the_order(U*V,X,R),
                eval_int(R,X,R1),ssimp(R1,Result).

int_by_parts(Expr,X,Result):-

                Expr =..[ln,X],Expr1 = Expr*1,
                eval_int(Expr1,X,R),ssimp(R,Result),!.

int_by_parts(Expr,X,R):- R = '"Sorry,I cannot integrate"',!.

/*-----------------------------------------------------------------*/

/*** select_the_order(Exp,x,R): orders the expression(Exp)
     to R ***/

select_the_order(U*V,X,R):-

                (
                 (
                  U=..[Op,_],
                  (Op==sin;Op==cos;Op==tan;Op==cot;Op==sec;
                                                   Op==cosec)
                 );
                 (
                  arg(1,U,U1),
                  (U1==e;const(U1,X)),
                  V =..[^,X,_]
                 );
                 (
                  arg(2,U,U2),
                  U2==2,U1=..[Op1,_],
                  (Op1==sec;Op1==cosec)
                 );
                 (
                  V=..[ln,_]);(arg(1,V,V1),
                  (
                   V1=..[ln,_];(V1==e,U =..[^,L,_],L\==X)
                  )
                 )
                ),
                R = V*U,!.

select_the_order(U*V,X,U*V):- !.

/*-----------------------------------------------------------------*/

/*** eval_int(Exp,X,R): evaluates the integral of ordered
     Exp to R ***/

eval_int(U*V,X,Result):-
```

```
                    stand_int(V,X,V1),!,simpl(U*V1,X,R),
                    L1 = [R],
                    diff(U,X,U1),simpl(U1*V1,X,Expr1),
                    test2_int(Expr1,X,U*V,L1,Newlist),
                    rev(Newlist,List),
                    eval_list(List,X,Result).

eval_int(U*V,X,Result):-

                    U =..[^,e,X],
                    test_arg1_arg2(V,X,R),!,
                    Result = U*R.

eval_int(U*V,X,Result):-

                    U =..[^,e,X],
                    V =..[/,V1,V2],
                    compare(<,V1,V2),
                    V2 =..[^,P,Q],
                    number(Q),
                    P =..[_,P1,P2],
                    atomic(P1),atomic(P2),
                    D = V1-P,sp(D,SimpD),
                    number(SimpD),
                    N is 1-Q,
                    Q1 is -Q,
                    NewV =..[+,P^(N),SimpD*P^(Q1)],
                    test_arg1_arg2(NewV,X,R),!,
                    Result = U*R.

eval_int(U*V,X,Result):-

                    U =..[^,e,X],
                    V =..[/,N,D],
                    (
                     D =..[Op1,1,cos(ArgD)];
                     D =..[Op1,cos(ArgD),1]
                    ),
                    NewArgD = 2^(-1)*ArgD,
                    simpl(NewArgD,X,SArgD),
                    (
                     (Op1=='+',NewD = 2*cos(SArgD)^2);
                     (Op1=='-',NewD = 2*sin(SArgD)^2)
                    ),
                    (
                    (N =..[Op2,Ln,sin(ArgD)],
                     number(Ln),
                     P = Ln*NewD^(-1));
                    (N =..[Op2,sin(ArgD),Rn],
                     number(Rn),
                     P = Rn*NewD^(-1))
```

```
                    ),
                    simpl(P,X,SimpP),
                    trig_simp(SimpP,X,SP),
                    (
                     (Op1=='+',Q = tan(SArgD));
                     (Op1=='-',Q = cot(SArgD))
                    ),
                    NewV =..[Op2,SP,Q],
                    test_arg1_arg2(NewV,X,R),!,
                    Result = U*R.

/*-------------------------------------------------------------*/

test2_int(-Expr,X,U*V,Oldlist,Newlist):-

                    append(Oldlist,[-],Oldlist1),
                    test2_int(Expr,X,U*V,Oldlist1,Newlist).

test2_int(Expr,X,U*V,Oldlist,Newlist):-

                    (
                     const_and_int(Expr,X,R1);
                     stand_int(Expr,X,R1)
                    ),
                    simpl(R1,X,R),
                    L = [R],
                    append(Oldlist,L,Newlist).

test2_int(Expr,X,U*V,Oldlist,Newlist):-

                    (
                    ((Expr==U*V;Expr==V*U),!,Const = 1);
                    (Expr =..[*,Expr1,Expr2],
                     const(Expr1,X),
                     (Expr2==U*V;Expr2==V*U),!,Const = Expr1)
                    ),
                    L=[Const],
                    append(Oldlist,L,Newlist).

test2_int(Expr,X,U*V,Oldlist,Newlist):-

                    Expr =..[*,C,NewExpr],
                    ((not const(C,X),!,E1=C,E2=NewExpr,C1=1);
                    NewExpr=..[*,E1,E2],C1=C),!,
                    select_the_order(E1*E2,X,NewE1*NewE2),
                    stand_int(NewE2,X,R),simpl(C1*NewE1*R,X,R1),
                    L = [R1],
                    append(Oldlist,L,L1),
                    diff(NewE1,X,D),simpl(C1*D*R,X,Expr2),
                    test2_int(Expr2,X,U*V,L1,Newlist).

/*-------------------------------------------------------------*/
```

```
/*** rev(L,M): reverses the order of elements in list L
     into list M ***/

rev([],[]).
rev([H|T],L):- rev(T,Z),append(Z,[H],L).

/*** append(X,Y,L): aapends the list X with list Y to
     new list L ***/

append([],L,L).
append([X|L1],L2,[X|L3]):- append(L1,L2,L3).

/*-------------------------------------------------------------*/

/*** eval_list(L,X,R): evaluates the list L to give
     result as R ***/

eval_list(List,X,Result):-

                hat(List,H1,T1),
                check_tail_eval(T1,X,H1,Result).

/*** hat(L,H,T): gives the head of list L as H and
     tail as T ***/

hat([H],H,[]):- !.
hat([H|T],H,T).

/*** check_tail_eval(List,X,H,R): checks the tail list (List)
     whose head value is H and evaluates the list to
     give final result as R ***/

check_tail_eval(T,X,H,Result):-

                hat(T,H1,T1),
                not const(H,X),
                H1\=='-',T1==[],Result =..[-,H1,H].

check_tail_eval(T,X,H,Result):-

                const(H,X),hat(T,H1,T1),
                (
                 (T1==[],R = H1);
                 (H1=='-',eval_list(T1,X,R));
                 (H1\=='-',eval_list(T,X,R))
                ),
                simp((1+H)^(-1),Const),
                Result = Const*R.

check_tail_eval(T,X,H,Result):-
```

```
                hat(T,H1,T1),
                H1\== '-',T1\==[],simp(H1,SimpH1),
                R =..[-,SimpH1,H],
                check_tail_eval(T1,X,R,Result).

check_tail_eval(T,X,H,Result):-

                hat(T,H1,T1),
                H1 == '-',R =..[-,H],
                check_tail_eval(T1,X,R,Result).

/*-------------------------------------------------------------*/

/*** test_arg1_arg2(Exp,X,R): tests 1st argument with the
     2nd argument of Exp to give result as R ***/

test_arg1_arg2(V,X,Result):-

               V =..[Op,V1,V2],
               (
               (Op=='+',
                (
                 (diff(V1,X,R1),compare(=,R1,V2),
                                    Result = V1);
                 (diff(V2,X,R2),compare(=,R2,V1),
                                    Result = V2);
                 (test1_int(V1,X,R3),compare(=,R3,V2),
                                    Result = V2);
                 (test1_int(V2,X,R4),compare(=,R4,V1),
                                    Result = V1)
                )
               );
               (Op=='-',
                (
                 (diff(V1,X,R1),compare(=,R1,-V2),
                                    Result = V1);
                 (diff(-V2,X,R2),compare(=,R2,V1),
                                    Result = -V2);
                 (test1_int(V1,X,R3),compare(=,R3,-V2),
                                    Result = -V2);
                 (test1_int(-V2,X,R4),compare(=,-R4,V1),
                                    Result = V1)
                )
               )
               ).

test_arg1_arg2(V,X,R):- !,fail.

/*-------------------------------------------------------------*/

differentiate :-readfile(file),
               repeat,
```

```
                differ,nl,nl,
   print('Do you want to differentiate another expression? '),
                read(Ans),
                nl,
                ((Ans\==yes,true);fail).

differ :-       nl,
   print('Give the expression to be differentiated :'),
                read(E),
                R =..[diff,E,x,Result],call(R),nl,nl,
                print('The differential of'),tab(3),
                print('"'),print(E),print('"'),
                tab(2),print('is = '),tab(2),
                print(Result),!.

/*-------------------------------------------------------------*/

diff(E,X,Result):- simp(E,SimpE),d(SimpE,X,R),simp(R,Result).

/*** d(Exp,x,R): gives differential of expression(Exp)
     w.r.to X as R ***/

d(X,X,1).
d(e^X,X,e^X):-!.
d(e^U,X,R):- d(e^U,U,R1),d(U,X,U1),R = U1*R1,!.
d(sin(X),X,cos(X)):-!.
d(cos(X),X,-sin(X)):-!.
d(tan(X),X,sec(X)^2):- !.
d(cot(X),X,-cosec(X)^2):- !.
d(cosec(X),X,-cosec(X)*cot(X)):- !.
d(sec(X),X,sec(X)*tan(X)):- !.
d(ln(X),X,X^(-1)):-!.
d(ln(U),X,R):- d(ln(U),U,R1),d(U,X,U1),R = U1*R1.
d(K,X,0):- const(K,X).
d(-T,X,-R):- d(T,X,R),!.
d(T,X,U1*R1):- T=..[F,U],U\==X,d(U,X,U1),d(T,U,R1),!.
d(U+V,X,U1+V1):- d(U,X,U1),d(V,X,V1),!.
d(U-V,X,U1-V1):- d(U,X,U1),d(V,X,V1),!.
d(K*U,X,K*U1):- const(K,X),d(U,X,U1),!.
d(U*V,X,U1*V+V1*U):- d(U,X,U1),d(V,X,V1),!.
d(X^N,X,N*X^M):- const(N,X),
                 (number(N),N\==0,M is N-1,!; M=N-1).
d(U^N,X,DU*R):- number(N),d(U^N,U,R),d(U,X,DU).
d(U^V,X,U1*V^U^(V-1)+ln(U)*U^V*V1):- not (const(V,X)),
                                    d(U,X,U1),d(V,X,V1),!.
d(U/V,X,R):- d(U*V^(-1),X,R),!.

/*-------------------------------------------------------------*/

/*** const(Exp,X): determines if expression(Exp) is a
     constant w.r.to X  or not */
```

```
const(Y,X):- atomic(X),atomic(Y),(X\==Y,!;!,fail).
const(X,X):-!,fail.
const(E,X):- E=..[_,Z],!,const(Z,X).
const(E,X):- arg(1,E,Left),!,const(Left,X),arg(2,E,Right),!,
             const(Right,X).
const(_,_).



/*-------------------------------------------------------------*/

/*** readfile(X): reads the file X ***/

readfile(X):-seeing(Old),see(X),readline(0),see(Old),!.

readline(Cr):-read_in(S,C),C1 is Cr+1,
        (((C==26;C==4),seen,!);
         (C1==23,nl,write('You want more?'),tab(2),
                seeing(Old),see(user),read(Ans),see(Old),
                ((Ans==yes,((nonvar(S),write(S));true),
                                        readline(0));
                 (seen,!)
                )
         );
         (var(S),nl,readline(C1));
         (write(S),nl,readline(C1))
        ).

read_in(W,C2):- get0(C),readword(C,W,C2).

readword(C,W,C2):- inword(C,NewC),!,get0(C1),
                   restword(C1,Cs,C2),name(W,[NewC|Cs]).
readword(C,W,C).

restword(C,[NewC|Cs],C2):- inword(C,NewC),!,get0(C1),
                           restword(C1,Cs,C2).
restword(C,[],C).

inword(C,C):- C>31,C<127.
inword(C,C):-C==9.
```

6. SESSION 1

```
?- integrate.

**********************************************************************
*                                                                    *
*                      // ADVICE TO THE USER //                      *
*                                                                    *
*       Always give the input expression in the cannonical form,     *
*                                                                    *
*       that is,                                                     *
*                                                                    *
*               [number]operator[constant]operator[f(x)].            *
*                                                                    *
*       examples: 3*a*sin(x),                                        *
*                 4+ln(3)+cos(x).                                    *
*                                                                    *
*       Any input must be terminated by a '.'(dot).                  *
*                                                                    *
*       Exponential operator is '^'.                                 *
*                                                                    *
********************** HAVE A NICE SESSION **************************

Give the expression to be integrated : x^5.

Integral of   "x^5"   is =   6^-1*x^6

Do you want to integrate another expression? yes.

Give the expression to be integrated : sin(x)+cos(4*x+3).

Integral of   "sin(x)+cos(4*x+3)"   is =   -cos(x)+4^-1*sin(4*x+3)

Do you want to integrate another expression? yes.

Give the expression to be integrated : a^x*cos(3*x+4).

Integral of   "a^x*cos(3*x+4)"   is =   (1+ln(a)^2*9^-2)^-1*(3^-1*(a^x
*sin(3*x+4))+ln(a)*9^-2*(a^x*cos(3*x+4)))

Do you want to integrate another expression? yes.

Give the expression to be integrated : ln(x).

Integral of   "ln(x)"   is =   ln(x)*x-x

Do you want to integrate another expression? yes.

Give the expression to be integrated : ln(x)/x.

Integral of   "ln(x)/x"   is =   2^-1*ln(x)^2
```

```
Do you want to integrate another expression? yes.

Give the expression to be integrated : e^x*(tan(x)-ln(cos(x))).

Integral of   "e^x*(tan(x)-ln(cos(x)))"   is =   e^x*(-ln(cos(x)))

Do you want to integrate another expression? yes.

Give the expression to be integrated : e^x*((1+sin(x))/(1+cos(x))).

Integral of   "e^x*((1+sin(x))/(1+cos(x)))"   is =   e^x*tan(2^-1*x)

Do you want to integrate another expression? yes.

Give the expression to be integrated : e^x*((x-1)/(x+1)^3).

Integral of   "e^x*((x-1)/(x+1)^3)"   is =   e^x*(x+1)^-2

Do you want to integrate another expression? yes.

Give the expression to be integrated : x^4*ln(x)^4.

Integral of   "x^4*ln(x)^4"   is =   5^-1*(ln(x)^4*x^5)-4*25^-2*(ln(x)
^3*x^5)+12*125^-3*(ln(x)^2*x^5)-24*625^-4*(ln(x)*x^5)+24*3125^-5*x^5

Give the expression to be integrated : cos(5*x+4)*x^6.

Integral of   "cos(5*x+4)*x^6"   is =   5^-1*(x^6*sin(5*x+4))+6*25^-2*
(x^5*cos(5*x+4))-30*125^-3*(x^4*sin(5*x+4))-120*625^-4*(x^3*cos(5*x+4)
)+360*3125^-5*(x^2*sin(5*x+4))+720*15625^-6*(x*cos(5*x+4))-720*78125^-
7*sin(5*x+4)

Do you want to integrate another expression? yes.

Give the expression to be integrated : sin(a*x+b)*x^5.

Integral of   "sin(a*x+b)*x^5"   is =   -a^-1*(x^5*cos(a*x+b))+5*a^-2*
(x^4*sin(a*x+b))+20*a^-3*(x^3*cos(a*x+b))-60*a^-4*(x^2*sin(a*x+b))-120
*a^-5*(x*cos(a*x+b))+120*a^-6*sin(a*x+b)

Do you want to integrate another expression? no.
```

7. SESSION 2

```
?- differentiate.

*****************************************************************************
*                                                                           *
*                        // ADVICE TO THE USER //                           *
*                                                                           *
*       Always give the input expression in the cannonical form,            *
*                                                                           *
*       that is,                                                            *
*                                                                           *
*               [number]operator[constant]operator[f(x)].                   *
*                                                                           *
*       examples: 3*a*sin(x),                                               *
*                 4+ln(3)+cos(x).                                           *
*                                                                           *
*       Any input must be terminated by a '.'(dot).                         *
*                                                                           *
*       Exponential operator is '^'.                                        *
*                                                                           *
************************* HAVE A NICE SESSION ******************************

Give the expression to be differentiated :cos(3*x).

The differential of   "cos(3*x)"  is =   -3*sin(3*x)

Do you want to differentiate another expression? yes.

Give the expression to be differentiated :x^7.

The differential of   "x^7"  is =   7*x^6

Do you want to differentiate another expression? yes.

Give the expression to be differentiated :a^x.

The differential of   "a^x"  is =   ln(a)*a^x

Do you want to differentiate another expression? yes.

Give the expression to be differentiated :x^x.

The differential of   "x^x"  is =   x^x^(x-1)+ln(x)*x^x

Do you want to differentiate another expression? no.
```

REFERENCES

1. W.F.Clocksin & C.S.Mellish, Programming in PROLOG, Springer-Verlag, Berlin, Heidelberg 1981.
2. Edited by Avron Barr & Edward A.Feigenbaum, The Handbook of Artificial Inte lligence, Vol I&II, William Kaufmann, Inc 1981.
3. Any standard text book on Calculus.
4. Toshinori Munakata, Procedurally Oriented Programming Techniques in Prolog, IEEE Expert, Summer 1986.