Informatics and Mathematical Modelling
Computer Science and Technology
Lasse Engbo Christiansen & Hans Bruun
Jan 2002

Contents

1. C++ for Those who Know Java, Project Problem January 2002

In C++ for those who know Java, you will learn C++ and you will have to make a big C++ program. This document describes the problem, which you are supposed to solve. The size of the project makes it necessary for you to work in a group of two or three students. It will also be an advantage to have some programming experience. How much of the project you are supposed to implement in groups of 2 or 3 is mentioned in section 5. The project and various programming techniques necessary to solve the problem will be explained during the lectures.

In the project problem described here, a system for expression manipulation has to be implemented in C++. In the sequel, the system will be called XWORKSHOP2002 (eXpressionWORKSHOP2002). In the system one can interactively – using commands – define arithmetic expressions, print, evaluate and transform expressions in a variety of ways, e.g. make symbolic differentiation. The systems command language is described in section 2 and section 3 shows a big example on using the XWORKSHOP2002 system. The description of the command language in section 2 is rather concise so it may be a good idea to read section 3 first or in parallel with the reading of section 2.

XWORKSHOP 2002

2. XWORKSHOP2002's Command Language

2.1 The Lexical Rules of the Command Language

The terminal symbols of the command language are shown below:

"**num**" , "**let**" , "**show**" , "**eval**" , "**diff**" , "**fold**" , "**subst**" , "**const**", "**quit**" , "**help**" , "**(**" , "**)**" , "**=**" , ":=" , "**+**" , "**–**" , "*" , "/" , ":".

Identifiers and numeric constants are also considered terminal symbols. In the sequel they are denoted by *Id* and *Const* respectively and are defined below:

Id : a sequence of letters and digits, beginning with a letter.
Const : a sequence of digits (the integer part), possibly the character "." and a (possibly empty) sequence of digits (the decimal part), which all together represent a decimal fraction.

The terminal symbols from "**num**" to "**help**" are called keywords. Besides terminal symbols a command may contain space characters. Spaces are only allowed between terminal symbols. Spaces are necessary to separate keywords and identifiers and are necessary to separate an identifier from a subsequent numeric constant.

A command must be on one line, i.e. a line shift character is not allowed in a command, but must follow the terminal symbol in the command. The terminating line shift character will initiate the execution of the command.

2.2 The Syntax of the Command Language

The syntax for the XWORKSHOP2002 command language is shown below using an EBNF-grammar. In this notation {X} means that X may be repeated zero ore more times. The semantic of the various language constructs in the command language is explained in the subsequent subsections.

```
Command   = DeclVar | DefExpr | ShowExpr | EvalExpr | Assign |
            "quit" | "help"
DeclVar   = "num" Id
DefExpr   = "let" Id "=" ME0
ShowExpr  = "show" ME0
EvalExpr  = "eval" ME0
Assign    = Id ":=" ME0

ME0     = E0 | MetaExpr
E0    = E1 { Pmoprt E1}
Pmoprt = "+" | "–"
E1    = E2 { Mdoprt E2}
Mdoprt = "*" | "/"
E2    =  UnOp "(" ME0 ")" | "(" ME0 ")" | Id | Const
UnOp = "sin" | "cos" | "exp" | "log"

MetaExpr    =    "diff" "(" Id ")" ME0 | "subst" "(" Id ":"ME0 ")" ME0 |
                 "fold" ME0 | "const" ME0
```

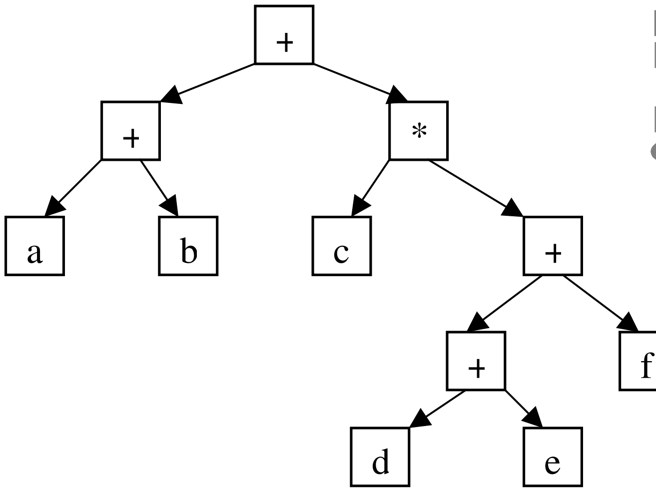
Operatorprioritet	
"+" , "-"	1
"*" , "/"	2
UnOp,( )	3

The operators have a relative priority as shown in the table to the right. This priority also appears implicitly from the grammar above. All operators are left associative.

2.3 Expressions

The sequences of terminal symbols that may be derived from *E0*, are called expressions. Expressions are used to specify symbolic expressions. The various commands in the command language can work with symbolic expressions in different ways: evaluate them, print them, differentiate them etc.

2.3.1 Symbolic Expressions



Usually one regards an expression as written text e.g. a+b+c*(d+e+f). In order to understand this expression correctly one has to know that * has greater priority than + and that + and * are both left associative. This may be specified explicitly by placing extra parentheses: (a+b)+(c*((d+e)+f)). Symbolic expressions are expressions regarded as a tree structure with operators in the nodes and operands in the leafs. The tree structure shows directly the operands of an operator as sub-trees, thereby avoiding the need for parentheses and concepts like priority and associativity. The tree structure shown to the right may be regarded as the symbolic expression for the textual expression shown previously.

2.3.2 Well-formed Expressions, From Textual Expressions to Symbolic Expressions

Expressions are a part of the majority of commands. How expression are written as text in a command is defined in the syntax for the command language, see section 2.2. A sequence of terminal symbols is a syntactically correct expression if it is derivable from `E0`, i.e. it is the syntax rules for `E0`, and the various forms of sub-expressions `E1`, and `E2`, which define the form of expressions. The textual expression, as it is written in the command, represents a symbolic expression, as defined in section 2.3.1. Below is a rule for each form, which a textual expression may have. In the rule for a textual expression derived from `E` we use the notation `S(E)` to denote the corresponding symbolic expression.

Expressions must be syntactically correct, but must furthermore satisfy some requirements in order to be well-formed. In section 2.7 about expression evaluation it is specified that the different expression operators expect operand values of some specific type, e.g. the operator + must have numeric operands.

2.3.3 Expression and Meta Expression

Commands contains ordinary expressions (derived from `E0`) as explained in section 2.3 and so-called meta-expression derived from `MetaExpr`. Meta-expression will be explained in section 2.9. Both ordinary expressions and meta-expressions denote a symbolic expression, but meta-expressions differ from ordinary expressions by obtaining the denoted symbolic expression by transformation of an other symbolic expression (e.g. symbolic differentiation). Ordinary expressions (derived from E0 – E2) have a direct equivalent symbolic expression whereas meta-expressions do not have an equivalent symbolic expression.

In some of the XWORKSHOP2002 commands working on a symbolic expression, this expression may be specified by either an ordinary expression or a meta-expression. This is indicated with the symbol `ME0` in the syntax for the command language (sect. 2.2). The symbol `ME0` is defined by the rule

```
ME0 = E0 | MetaExpr
```

For a `ME0`-expression of the form `E0` holds *S*(`ME0`)= *S*(`E0`) and for a `E0`-expression of the forme `MetaExpr` holds *S*(`ME0`)= *S*(`MetaExpr`).

2.4 Variable Declaration

When running the expression system XWORKSHOP2002, the system allocates storage of places. A place may contain a numeric[1] variable. By use of different commands, the user may store values in and retrieve values from these places.

DeclVar = "**num**" *Id*

The execution of a *DeclVar* command of the form "**num**" *Id* results in *Id* being declared as a numeric variable. This has two consequences:

- A new place – corresponding to Id – is allocated in the storage. This place contains a numeric value and it is initialised to the value 0.
- The identifier *Id* is defined to denote the symbolic expression NVar(*Id*). In expressions in all the subsequent commands, *Id* will denote this symbolic expression as described in section 2.3.2 (*E2*-expression).

No subsequent command may re-declare or re-define the identifier *Id*.

2.5 Expression Definition

- *DefExpr* = "**let**" *Id* "=" *ME0*

A *DefExpr* command has the form "**let**" *Id* "=" *ME0* . The execution of the command results in *Id* being defined to denote the symbolic expression *S(ME0)*. The definition is valid for the remaining part of the session or until the identifier *Id* in a subsequent command is declared as a variable or is re-defined. The execution of the command furthermore results in the text *Id = unpars(S(ME0))* being written on the next line. Here *unpars(S(ME0))* is a textual representation of the symbolic expression *S(ME0)*. See section 2.6.

You should notice that after this command, *Id* is the name of a symbolic expression, so *Id* is *not* a variable. Furthermore, note that both when *Id* is declared as a variable (in a *DeclVar* command) and when *Id* is defined to be the name of a symbolic expression (in a *DefExpr* command) *Id* denotes a symbolic expression. When *Id* is declared as a variable in a DeclVar-command the command execution moreover results in the allocation of a variable having this name.

2.6 Showing an Expression

- *ShowExpr* = "**show**" *ME0*

A *ShowExpr* command has the form "**show**" *ME0*. When executed the command prints on the next line the text *unpars(S(ME0))*. Here *unpars(S(ME0))* is a textual representation of the symbolic expression *S(ME0)*. This textual representation is characterised by having the minimal number of parentheses for showing the structure of *S(ME0)* using the given priority and associativity of the operators in the expression.

2.7 Expression Evaluation

- *EvalExpr* = "**eval**" *ME0*

A *EvalExpr* command has the form "**eval**" *ME0*. When executing the command the value of the symbolic expression *S(ME0)* is calculated. The calculated value is written on the next line.

The evaluation of a symbolic expression – with no errors – results in a numeric value. If the evaluation of an expression results in an error, an error message is written, and the result of the erroneous sub-expression becomes 0.

---

[1] In these notes the word numeric is used to mean an integer or a real value.

2.8 Assignment

- *Assign = Id* “**:=**” *ME0*

An *Assign* command has the form *Id* “**:=**” *ME0*. For the command to execute correctly the identifier *Id* must have been declared as a *variable* in a previous variable declaration; otherwise the system writes an error-message and the assign command is stopped. The command evaluates the symbolic expression *S(ME0)* which results in a value *ev*. The value *ev* is stored in the place for *Id* in the storage.

2.9 Meta-expression

In this section, the various forms of meta-expressions will be explained. A meta-expression denotes a symbolic expression just as an ordinary expression. But meta-expressions differ from ordinary expression by obtaining the denoted symbolic expression by transformation of an other symbolic expression. The possible transformations are symbolic differentiation, simplification and substitution.

2.9.1 Differentiation

- *MetaExpr =* “**diff**” “**(**” *Id* “**)**” *ME0*

A *MetaExpr* of the form “**diff**” “**(**” *Id* “**)**” *ME0* denotes the symbolic expression, obtained by differentiating the symbolic expression *S(ME0)* with respect to the variable denoted by *Id.* The identifier *Id* must be declared as a numeric variable in a previous command.

2.9.2 Substitution

- *MetaExpr =* “**subst**” “**(**” *Id* “**:**”*ME0* “**)**” *ME0*

For a *MetaExpr* of the form “**subst**” “**(**”*Id* “**:**”$ME0_1$ “**)**” $ME0_2$ it is required that *Id* has been declared as a variable. The Meta-expression denotes a symbolic expression derived from $S(ME0_2)$ in the following way:

All sub-expressions in $S(ME0_2)$ of the form Nvar(*Id*) are replaced by $S(ME0_1)$.

2.9.3 Converting to Constant Expressions

- *MetaExpr =* “**const**” *ME0*

A *MetaExpr* of the form “**const**” *ME0* denotes the symbolic expression obtained from *S(ME0)* in the following way. Let *v* be the value obtained by evaluating the symbolic expression *S(ME0)*, then the new symbolic expression is Nconst(v);

2.9.4 Simplification of Expressions

- *MetaExpr =* “**fold**” *ME0*

A *MetaExpr* of the form “**fold**” *ME0* denotes the symbolic expression, which is obtained by simplifying the symbolic expression *S(ME0)* as much as possible by use of a set of rules partially shown in the sequel.

Rules for the commutative and associative law for + and *:

Add(e1, e2) = Add(e2, e1)
Mul(e1, e2) = Mul(e2, e1)
Add(e1, Add(e2, e3)) = Add(Add(e1,e2),e3)
Mul(e1, Mul(e2, e3)) = Mul(Mul(e1,e2),e3)

Rules about the neutral elements 1 and 0:

Add(Nconst(0), e1) = e1
Mul(Nconst(1), e1) = e1
Div(e1, Nconst(1)) = e1
Sub(e1,Nconst(0)) = e1

Rules for 0 in connection with * and / :

Mul(Nconst(0), e1) = Nconst(0)
Div(Nconst(0), e1) = Nconst(0)

Rules about signs:

Sub(e1, Add(e2, e3)) = Sub(Sub(e1,e2),e3)

Evaluation of constant expressions:

Add(Nconst(r1), Nconst(r2)) = Nconst(r1+r2)
Sub(Nconst(r1), Nconst(r2)) = Nconst(r1-r2)
Mul(Nconst(r1), Nconst(r2)) = Nconst(r1*r2)
Div(Nconst(r1), Nconst(r2)) = Nconst(r1/r2)

Exp(Nconst(r))= Nconst(e^r)
Log(Nconst(r)) = Nconst(log(r))
Sin(Nconst(r)) = Nconst(sin(r))
Cos(Nconst(r)) = Nconst(cos(r))

These are some rules that can be used simplifying expressions.

2.10 quit and help

The quit-command stops XWORKSHOP2002. The help-command prints simple instructions on how to use XWORKSHOP2002.

3. Example of Use

`XWORKSHOP 2002`	Message from XWORKSHOP2002, > is the system prompt. XWORKSHOP2002 is now waiting for a command from the user.
`>num x`	x is *declared* as a numeric variable.
`>num y`	y is *declared* as a numeric variable.
`>eval x` `0`	Evaluate the expression, which is just the variable x. The content of the variable has automatically been initialised to the value 0.
`>bool s`	s is declared as a logic variable.
`>eval s` `false`	Evaluate s. Because s is declared as a logic variable it is initialised to false.
`>x:= 5.3`	The value 5.3 is stored in the variable x.
`>x:= x+ 2`	The value x+2 is stored in the variable x,
`>eval x` `7.3`	so x now contains the value 7.3
`>let a= x+y+2` `a= x+y+2`	Here a is *defined* to denote the symbolic expression corresponding to `x+y+2` . I.e. a is *not* a variable as x and y, but is just the name of a symbolic expression.
`>let b= a*(3.5+a)` `b= (x+y+2)*(3.5+x+y+2)`	Here b is *defined* to denote another symbolic expression, which is constructed from the other symbolic expression denoted by a. Note that the system on line after the command prints a textual representation of the symbolic expression which b denotes. This textual representation has just the necessary parentheses.
`>show a*b` `(x+y+2)*(x+y+2)*(3.5+x+y+2)`	The show-command prints a textual representation of the symbolic expression, which a*b denotes. Note the added parentheses.

```
>let c=x-y
c=x-y
```

Here c is defined.

```
>show 2-c
2-(x-y)
```

When printing the textual representation of a symbolic expression the necessary parentheses are added,

```
>show c-2
x-y-2
```

but only the necessary parentheses.

```
>eval a
9.3
```

Evaluation of the symbolic expression which a denotes. The variable y is part of this expression,

```
>y:= 2.7
```

so if we change the value of the variable y,

```
>eval a
12
```

the evaluation of a also yields a new result.

```
>let sincosx= sin(x)+cos(x)
sincosx= sin(x)+cos(x)
```

In expressions the functions sin, cos, exp and log may be used.

```
>let escx= exp(x)*sincosx
escx=exp(x)*(sin(x)+cos(x))
>x:= 0
>eval escx
1
>x:= 1
>eval escx
3.75605
num z
>show a
x+y+2
```

We now try the more interesting part of XWORKSHOP2002, namely the *meta-expressions,* i.e. expressions, which yield a new symbolic expression obtained by transforming a given symbolic expression.

```
>let dax= diff(x)a
dax= 1+0+0
```

We start by computing the symbolic expression which is the differential coefficient of a with respect to x. Diff only knows about differentiation, so the result does not look very professional.

```
>let fdax= fold dax
fdax= 1
```

Fortunately, we have the fold-meta-expression, which knows something about reducing expressions and evaluation of constant expressions.

```
>show b
b=(x+y+2)*(3.5+x+y+2)
>let dbx= diff(x)b
dbx=(x+y+2)*(0+1+0+0)+(3.5+x+y+2)*(1+0+0)
```

Using fold here might be a good idea.

```
>let fdbx= fold dbx
fdbx= 7.5+x+y+x+y
```

Great, but note that fold does not know that x+x looks nicer if written as 2*x.

```
>let h= sin(2*exp(x))
h= sin(2*exp(x))
```

Let's try a more difficult expression.

```
>let dhx= diff(x) 3*h
dhx= 3*cos(2*exp(x))*(2*exp(x)*1+exp(x)*0)+sin(2*exp(x))*0
>let fdhx= fold dhx
fdhx= 6*cos(2*exp(x))*exp(x)
>let fdhx1= fold diff(x) 3*h
fdhx1= 6*cos(2*exp(x))*exp(x)
```

Note, it is not necessary to make the transformations one by one; you can make them all in one step.

Now, it only remains to try substitution:

```
>num z
```

Now we have three numerical constants: x, y and z.

```
>let a= sin(x+y+z)
a= sin(x+y+z)
>let dx= diff(x) log(x+1)*a*a
```

A complicated expression (depending on both x, y and z) is here differentiated with respect to x.

```
dx=
log(x+1)*sin(x+y+z)*cos(x+y+z)*(1+0+0)+sin(x+y+z)*(log(x+1)*cos(x+y+z)*(1+0+0)+sin(x+y+z)*1/(x+1)*(1+0))
>let fdx= fold dx
fdx=log(1+x)*sin(x+y+z)*cos(x+y+z)+sin(x+y+z)*(log(1+x)*cos(x+y+z)+sin(x+y+z)/(1+x))
>x:= 0
```

We would now like to study the result as a function of only y and z for fixed x=0:

```
>show fdx
log(1+x)*sin(x+y+z)*cos(x+y+z)+sin(x+y+z)*(log(1+x)*cos(x+y+z)+sin(x+y+z)/(1+x))
```

But assigning 0 to x (x:= 0) does not change the symbolic expression at all, but of course it will evaluate to a different value.

```
>let fdx0= subst(x:0) fdx
```

What should be done, is to substitute the variable x with the numerical constant 0.

```
fdx0=log(1+0)*sin(0+y+z)*cos(0+y+z)+sin(0+y+z)*(log(1+0)*cos(0+y+z)+sin(0+y+z)/(1+0))
>let ffdx0= fold fdx0
ffdx0=sin(y+z)*sin(y+z)
```

Now it is manageable.

And we can also do it all in one big step:

```
>let ffdx0= fold subst(x:0) fold diff(x) log(x+1)*a*a
ffdx0=sin(y+z)*sin(y+z)
>quit
goodbye
```

This was a small presentation! Try yourself!

4. Detection af Errors

It is important that XWORKSHOP2002 reacts sensible to errors and makes understandable error-messages, here are some examples:

```
Expression Shop 2002
```

.

```
>num x
```

x is declared as a numeric variable.

```
>num y
```

y is declared as a numeric variable.

```
>num x
x is already declared as a variable
```

Once an identifier is declared as a variable its meaning cannot be changed.

```
>let x = 2
x is declared as a variable
>let a= 2
a= 2
>let a= a+5
a= 2+5
```

Note that it is legal to change a *defined* identifier.

5. Minimum requirements and proposals for extensions

A group should – independent of its size – be able to implement all the commands in the XWORKSHOP2002 command-language, described in section 2.2 , including all normal expressions (derived from `E0` – `E2`) and the meta-expression “**`diff`**” “**`(`**” `Id` “**`)`**” `ME0` for differentiation, the meta-expressions “**`subst`**” “**`(`**” `Id` “**`:`**”`ME0` “**`)`**” for substitution, “**`const`**” `ME0` for converting to constants, and a simple “**`fold`**”, which takes care of some of the simpilfications mentioned above. A group with three members should also be able to implement a more enhanced version of the fold meta-expression, as in collecting constants, and a power function should also be included. Overall, good error-detection and good error-messages counts.

After this, try to extent the expressions of the command-language such that exponentiation becomes possible. You decide – by yourself – the detailed syntax (including priority and associativity). Furthermore, try to extent the meta-expressions of the command-language with other possible transformations of symbolic expressions. Among the possibilities we mention other simplifications from the more commonplace as x+x=2x to simplifications, which require knowledge about formulae for cos, sin, exp and log. Another possibility is integration (indefinite integrals). This is in general not possible, so the system must be able to recognise sub-expression, where specific integration methods may be applied (difficult). A source for inspiration could be well known symbolic computer algebra packages like e.g. Maple and your old textbooks about mathematical analysis.

6. Report Requirements

The report should contain sections as mentioned below:

- **Problem Analyses and Description of Problem Solution**. A short analysis of the problem and a description of the problem actually solved. Furthermore, a statement of possible ambiguities found in the problem formulation and a description of decisions made to repair these ambiguities.
  The description of the problem solution includes a description of how you have implemented the XWORKSHOP2002-systemet. This consists of
  - The main classes, their methods/functions and how the classes interact.
  - The algorithms and data-structures, which have been used.
- **Source code**: The source code with appropriate comments. The code must be available on A4-paper.
- **Testing/debugging**: Show the main examples used in the test and explain the extent to which these examples have tested all the various parts of the program.
- **Users Guide**: Should only contain changes compared to the given command-language as described in section 2.

Note: In this course you will be evaluated mainly from the program you have developed and not so much from the quality of the report. But, it is essential, that the report documents what has been made and documents that the constructed system actually works.
Note: A diskette containing the constructed program and source code should be attached to the delivered report.

January 2002
Lasse Engbo Christiansen.