An introduction to programming through C++
Ch. 24 : Structural recursion
Layout of mathematical formulae

Abhiram Ranade

Computers can do arithmetic. What about algebra?

Can computers be made to handle algebraic expression?

Can computers compute symbolic derivatives/integrals?

Commercial programs such as Mathematic can process algebra

Public domain programs such a TEX can typeset math formulae

In this chapter we study

- How to represent algebraic expressions
- How to produce nice layouts (example soon)
- Our representation and ideas will be useful for doing algebra..

The problem

Input: A textual description of a formula e.g.

```
\pi = \cfrac{4}{1+\cfrac{1^2}{3+\cfrac{2^2}
{5+\cfrac{3^2}{7+\cfrac{4^2}{9+\ddots}}}}}
```

Output: A visually pleasant layout of it:

$$\pi = \cfrac{4}{1+\cfrac{1^2}{3+\cfrac{2^2}{5+\cfrac{3^2}{7+\cfrac{4^2}{9+\ddots}}}}}$$

Overview

- How will the user describe the formula to a computer.
- How will the formula be represented in the memory of the computer.
- How will we determine the position and the sizes of the different parts of the layout.

A language to describe mathematical formulae

- Many languages already available. e.g. TEX, which was used to describe the formula for π shown earlier.

  Very elaborate.

- C++ expression syntax is also adequate for many kinds of expressions.

  Simpler than TEX, but still tricky.

Our choice:

- Allow only $+$ and $/$ operators.
- Specify as in a C++, but parenthesize every operator.

  "Fully parenthesized C++ expression."

- For simplicity operands must consist of a single alphanumeric character.
- For simplicity no spaces inside the description.
- Eliminating these restrictions: exercises.

Examples of output and input

	Desired output	Input required by our program
0.	a	`a`
1.	$\frac{a}{b+c}$	`(a/(b+c))`
2.	$a+\frac{b}{c}$	`(a+(b/c))`
3.	$a+b+c+d$	`(((a+b)+c)+d)`
4.	$\frac{x+1}{x+3}+\frac{x}{5}+6$	`((((x+1)/(x+3))+(x/5))+6)`

Structure of a mathematical formula

A math formula as we have represented can be thought of as having one of the following forms

- The formula is a single alphanumeric characteral.
  x "Primitive Formula"
- The formula is a sum, i.e. has the form
  (smaller formula 1 $+$ smaller fomula 2)
- The formula is a ratio, i.e. has the form
  (smaller formula 1 $/$ smaller fomula 2)

Non primitive formulae contain inside themselves other formulae!

Thus formulae have heirarchical/recursive structure.

Representing a formula on a computer

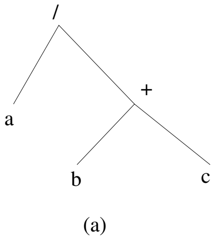
"Obvious" solution: The text specified by the user is itself a representation.
Drawback: It is difficult to extract parts of the formula.

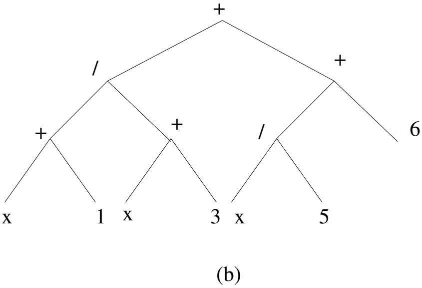
Important principle: Think of trees when representing anything with a heirarchical/recursive structure.

Our representation:

- Primitive formulae are represented by a single node.
- Sum formulae are represented by a tree: a root node associated with a label "+", left subtree consisting of the representation of the left summand, right subtree consisting of the representation of the right summand.
- Ratio formulae are similarly represented: root node associated with a label "/", left subtree consisting of the representation of the numerator, right subtree consisting of the representation of the denominator.

Examples





(a) Tree for $\frac{a}{b+c}$ (b) Tree for $\frac{x+1}{x+3} + \frac{x}{5} + 6$

Representation in a program

```
struct Node{
  char op;       // operator associated with node if any
  string value; // value associated with node, if any
  Node* L;       // pointer to left subformula, if any
  Node* R;       // pointer to right subformula, if any
  Node(char op1, Node* L1, Node* L2){
    op = op1;
    L  = L1;
    R  = R1;
  }
  Node(string v){
  // simplified constructor for primitive formulae
    value = v;
    op = 'P';    // 'P' denotes primitive formula
    L = R = NULL // No subformulae.
  }
  // other member functions to be described later
}
```

Creating formulae inside a program

```
Node aexp("a"), bexp("b"), cexp("c");
Node bplusc('+', &bexp, &cexp);
Node f1('/', &aexp, &bexp)

Node f2 = new Node('/', new Node("a"),
                   new Node('+', new Node("b"),
                                 new node("c")));
```

Reading in formulae from the keyboard

Implication of recursive structure:

- If first character read is an alphanumeric character, then the user must be typing a primitive formula, which is given by the character read.
- If the first character read is '(', then the user must be typing in a sum or a ratio. So we expect that following the '(' the user will type
  1. A formula representing the summand or the numerator. To read this in, we simply recurse.
  2. The operator, which will be a character '+' or '/'.
  3. Another formula, representing the summand or denominator. To read this in, we simply recurse.
  4. The character ')' signifying the end of the sum or the ratio.

We can put the code for reading in the formula into a constructor.

```
int main(){
  Node formula(cin);  // Described next.  Argument gives
  // the stream from which to read in the formula.
}
```

The code

```
Node::Node(istream &infile){
  char c=infile.get();
                              // is it a primitive formula?
  if((c >= '0' && c <= '9') ||
     (c >= 'a' && c <= 'z') ||
     (c >= 'A' && c <= 'Z')){
    L=R=NULL; op='P'; value = c;
  }
  else if(c == '('){       // is it a non-primitive formula?
    L = new Node(infile); // recursively get the L formula
    op = infile.get();    // get the operator
    R = new Node(infile); // recursively get the R formula
    if(infile.get() != ')')
      cout << "No matching parenthesis.\n";
  }
  else cout << "Error in input.\n";
}
```

The layout algorithm: outline

When we draw the layout, we must specify where to draw it.

```
void Node::draw(double x, double y){ // top left corner
  switch(op){
  case 'P':
    Text(x + textWidth(value)/2,
         y + textHeight()/2, value).imprint();
    break;
  case '+':  ... How do we do this? ...
  case '/':  ... How do we do this? ...
  default: cout << "Invalid input.\n";
  }
}
```

To layout a sum formula, we must know how the summands must align with each other.
To layout a ratio formula, we must know how long to draw the line denoting the division.

Important geometric features of a formula

Width: If we are laying out $(f + g)$ we must know the width of formula f to decide where to draw the $+$ symbol.
Height: If we are laying out (f/g) we must know the height of f to decide where to draw the horizontal bar.

Is this enough?

Example: $f = \dfrac{2}{\dfrac{1}{a} + b}, \quad g = a.$

$$\frac{2}{\dfrac{1}{a} + b} + a$$

Descent: Extent (distance) to which a formula dips below the horizontal line of the $+$
Ascent: Extent to which a formula rises above the horizontal line of the $+$.

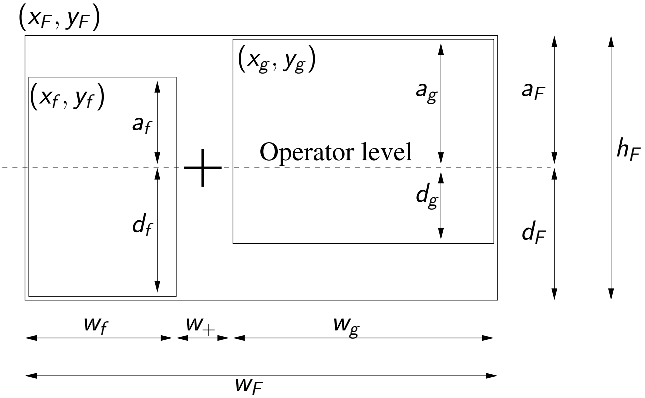
Ascent, descent are not independent: ascent $+$ descent $=$ height.

Parameters width, height, ascent can be determined recursively.

Recursively determining the parameters

Suppose $F = (f + g)$.

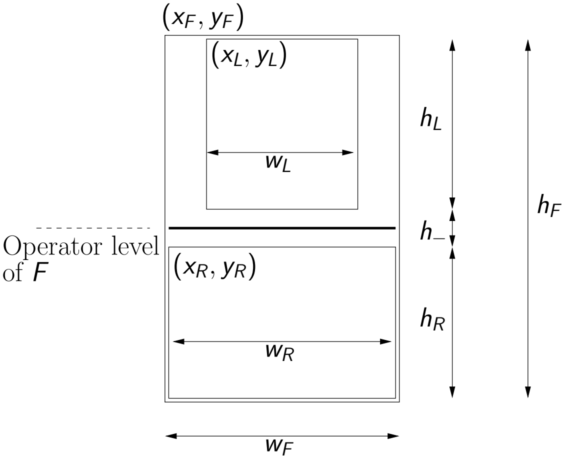
Parameters of F and f, g are related as below



Recursively determining the parameters - 2

Suppose $F = (f/g)$.

Parameters of F and f, g are related as below



The final code: definition of Node

```
struct Node{
  static const int h_bar = 10; // space for horizontal bar
  Node *L, *R;
  char op;
  string value;
  double width, height, ascent, descent;
  Node(string v);
  Node(char op1, Node* L1, Node* R1);
  Node(istream& infile);
  void setSizes();
  void draw(double clx, double y); // to actually draw
};
```

Code to set parameters: function setSizes

```
void Node::setSizes(){
  switch (op){
  case 'P':                        // Primitive formula
    width = textWidth(value);
    height = textHeight(); ascent = descent = height/2;
    break;
  case '+':                          // case L+R
    L->setSizes();
    R->setSizes();
    width = L->width + textWidth(" + ") + R->width;
    descent = max(L->descent, R->descent);
    ascent = max(L->ascent, R->ascent);
    height = ascent + descent;
    break;
  case '/':                         // case L/R
  ...
  }
}
```

Code to set parameters: function setSizes

```
void Node::setSizes(){
  switch (op){
  ...
  case '/':                     // case L/R
    L->setSizes();
    R->setSizes();
    width = max(L->width, R->width);
    ascent = h_bar/2 + L->height;
    descent = h_bar/2 + R->height;
    height = ascent + descent;
    break;
  default: cout << "Invalid input.\n";
  }
}
```

Code for drawing

```
void Node::draw(double x, double y){
  switch(op){  // case 'P' given earlier
  case '+':
    L->draw(x, y + ascent - L->ascent);
    R->draw(x + L->width + textWidth(" + "),
            y + ascent - R->ascent);
    Text(x + L->width + textWidth(" + ")/2, y + ascent,
         string(" + ")).imprint();    // draw the '+'
    break;
  case '/':
    L->draw(x + width/2 - L->width/2, y);
    R->draw(x + width/2 - R->width/2,
            y + L->height + h_bar);
    Line(x, y + ascent, x + width, y + ascent).imprint();
    break;
  default: cout << "Invalid input.\n";
  }
}
```

Concluding Remarks

- Formulae have a recursive representation.
- Recursion can be used to determine layout parameters.
- Once you represent formulae, you can manipulate them, e.g.
  evaluate them given the values of variables,
  take their derivatives with respect to a variable,
  find the product of formulae.