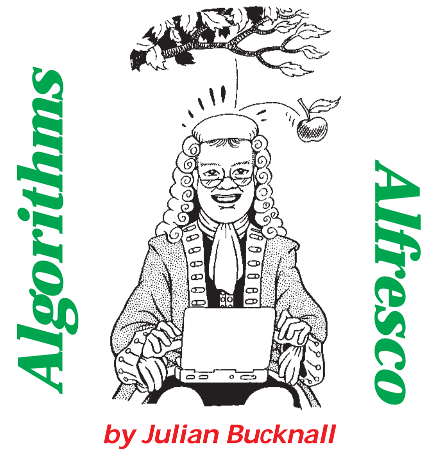
Occupy Your Mind

We take a look at parsing, reverse Polish notation, and cheating at homework



by Julian Bucknall

A couple of months ago I came home from work to find my wife and her 10-year-old brother sitting on the floor surrounded by pieces of paper with writing on. It transpired that he had some maths homework to do: using all of the digits 2, 3, 5 and 7, and the normal arithmetic operators, in any combination, write down expressions that evaluate to all of the numbers from 1 to 100. For example, 2+3+5+7=17; that's one of the numbers ticked off, only 99 left to go.

The homework was obviously an exercise in aiding mental arithmetic, and after some messing around with pen and paper helping out and finding some answers, I got to thinking of how I might write a program to do it. Unfortunately for Jeremy I didn't finish it that evening (I had other things to do) and so he had to rely on brain power, which was better for him anyway.

Reputation

The answer I came up with recalled a none too impressive part of my degree years. When I was a lad, doing my mathematics degree at King's College, London, we had the opportunity of taking a couple of programming courses as part of the degree. Not on your modern PCs, I hasten to add: in those days the computer department at KCL had a timeshare to a large mainframe for the whole of the University of London. You wrote your programs in FORTRAN on punched cards (there were six punch machines in the computer department), submitted them and, if you were lucky, got them back three hours later with the printout from hell. Analyze the results, fight to get back on the punch machines, resubmit, get the results back hours later, etc, etc. All in all, a pretty vile introduction to programming. Heaven knows why I was bitten enough with the programming bug to continue and make a career out of it.

Anyway, I managed to get through the introductory programming course (I wrote a program to generate 4x4 magic squares), and the next term I started the advanced course. I can't remember now the requisites of the course, but the end of course project had to be fairly complex. I decided to write a program to differentiate algebraic expressions. Complete lunacy. Armed with Knuth's *The Art of Computer Programming, Volume 1*, and some FORTRAN programming book that told me about parsing expressions, I made a valiant attempt and failed. I just could not get the program to read and process text properly. A real mess: I was spending 15 hours a week just on this project, the rest of my work was suffering, and so I decided to abandon the course. (With hindsight, I think the problem was that the particular flavour of FORTRAN I was using read text two characters at a time into a computer word, and I was processing the whole word as a single character.)

So this time around in *Algorithms Alfresco*, we're going to look at expression parsing, reverse Polish notation, expression evaluation (including Jeremy's homework) and, finally, put my course project to rest and discuss differentiation.

Little By Little

So the first problem is to understand an expression like (3+5*2)-4*2; in other words to recognize the individual operands and operators (this is called *parsing* the expression). The second problem is then to evaluate it.

Usually the way to parse an expression is to write a *recursive-descent parser*, but just for fun we'll take an alternative route using a stack. What we shall do is to create an RPN version of the expression (RPN stands for *Reverse Polish Notation* after the Polish logician, Jan Lukasiewicz). The reason for this is threefold. Firstly, the RPN version of an expression does not use parentheses. Secondly, we don't have to worry about precedence of operators (this follows on from the lack of parentheses). Thirdly, evaluating an RPN expression is simplicity itself. So, what does an RPN expression look like? Those of you who have used the old-style Hewlett-Packard calculators are intimately familiar with RPN and could convert our example expression into RPN at the drop of a hat. For the others, (3+5*2)-4*2 is 352*+42*- in RPN. The reason for the 'reverse' part of the name RPN is that the operator always appears *after* the operands it uses.

Evaluating an RPN expression requires the use of a stack (those old Hewlett-Packard calculator users are now starting to squirm a little as it all comes back). The stack holds operands and results of previous calculations. Let's evaluate our example RPN expression (352*+42*-) so you can see how easy it is. We read the expression from the left to the right. Take the 3 and push it on the stack. Take the 5 and do likewise. Similarly for the 2. Now we have a * operator. This requires two operands, so pop two off the stack, the 2 and the 5, apply the multiply operator to give 10 and push the result onto the stack again. Next we have the + operator. Again two operands are required, we pop off the 10 and the 3 this time. Add them (13) and push the result onto the stack.

What's next? We have a 4 (push it) and a 2 (push it). Now there's a * operator, so we pop off two operands from the stack (the 4 and the 2), multiply them (8) and push the answer onto the stack. Next in line is the final - operator. If you'd been paying attention, you'd know there are two items on the stack, first to pop being an 8 and the next a 10. Subtract them to give 2 and push the result onto the stack. We've run out of expression string at this point, and the result is the only thing left on the stack: 2. I'm sure you'll agree that writing an evaluator for an RPN expression is pretty simple, both to understand, and, as we'll see in a moment, to code.

But. (And it's a big but.) The first problem remains: to parse a normal algebraic expression and create an RPN version of it. Our example expression is relatively complex: it has parentheses and requires knowledge of operator precedence. Note that throughout this article we shall use Pascal's precedence rules: parentheses are the most important, followed by unary minus, or negative, followed by multiply and divide, followed by add and subtract. Exponentiation, which Pascal does not support, will be slipped in between unary minus and multiply/divide. A unary operator has one operand, a binary operator has two.

In Private

So let's start reading the expression and decide what to do about each of the components (or *tokens*) in the expression. For this first exposure to parsing, we'll assume that each token is one character. The first character is a left parenthesis. Um, don't know what to do with *that* just yet, so we'll push it onto a stack and get it out of the way (I warn you, this article is going to be Stack City). What's next? The digit 3. Not quite sure what to do with that one either. Since the RPN expression we are trying to build has just operators and operands, we'll use two stacks during our parsing: one for operators and one for operands. So push the 3 onto the operand stack. Right, onwards; the next token is a + sign. We don't know yet what to add to what, so just push it onto the operator stack. So far, it's been most boring.

And next? We read the 5 operand. We could pop off the top operator and the top operand, and combine them with the 5 to make 35+. But we won't do that *just* yet, we'll defer the clever stuff for a moment. What's next? The * operator.

This is where it gets interesting. The rule I shall quote (and apply) is going to be this: if the operator at the top of the operator stack is of *lower* or *equal* precedence, then push the current operator token onto the operator stack. On the other hand, if the operator at the top of the stack is of *greater* precedence, then pop it and two operands, combine them into an RPN expression and push the result onto the operand stack.

Now we'll decide what to do with the operator token we have in hand, obeying the rule I just quoted. In our case, the * operator has greater precedence than the + operator on the top of the stack so we just push it.

Now it's the right parenthesis. Time for another rule. If we get a right parenthesis we start popping operators and operands off the stacks and combining them into RPN expressions and pushing the results onto the operand stack (mimicking the algorithm to evaluate an RPN expression in fact), until we pop off the left parenthesis. If we never pop off a left parenthesis, the original expression was badly formed and we can signal an error.

At this point in our example, the operator stack looks like this, in a sideways representation with spaces separating the elements:

```
( + *
    ^
```

with the top of the stack marked with a caret. The operand stack looks like this:

```
3 5 2
    ^
```

Pop off the * operator. It's a binary operator and requires two operands, so pop off two of them (the 2 and the 5), combine them in the RPN format (52*) and push the answer onto the operand stack. Now, pop off the + operator, pop off the two top operands (52* and 3) and combine them into the RPN form (352*+) and push onto the operand stack. Pop off the top operator, which is the left parenthesis, and we're done with that particular operation.

Onwards. The next token from the expression is a - operator. Push it onto the operator stack. The next token is a 4. Push it onto the operand stack. The next is a * operator: we look at the top of the operator stack, it's a -, which is of lesser precedence, and so we push the *. Next is a 2, which is pushed onto the operand stack. Next is nothing at all: we've parsed the entire expression. What happens next is that we 'pretend' the end of the expression is like having a right parenthesis and pop off and combine operators and operands until there is only one operand left (we hope). The operator stack looks like this:

```
- *
  ^
```

and the operand stack like this:

```
352*+ 4 2
        ^
```

Pop off the * operator and two operands, combine to give 42* and push onto the operand stack. Pop off the - operator and two operands, combine to give 352*+42*- and push onto the operand stack. And that's it, the operator stack is empty, with only one element on the operand stack, which is the RPN expression we wanted. Phew!

Although going through this example, step by laborious step, might seem long-winded, it is actually very simple. It uses no recursion, unlike its better known stable-mate, the recursive descent parser *[and we all know how much Julian hates recursion! Ed]*.

It can detect errors in the original expression string quite easily. I mentioned one error, a right parenthesis without a corresponding left parenthesis, but others are found just as easily. If there is only one operand left for a binary operator such as multiply then we have an error. If we are left with a left parenthesis on the operator stack, then we have an error. And so on.

One thing I haven't talked about is the unary operators like plus and minus. How do we recognize those? Unfortunately, detecting them causes the parsing algorithm to lose some of its elegance, and is probably why the recursive descent parser is better known and used. The problem with a unary minus (negative) compared with a binary minus (subtraction) is that *they are the same character!* We have to separate the two uses by the context in which they're found.

A unary operator is found immediately *after* another, binary, operator or a left parenthesis, and immediately before an operand or a left parenthesis. That's all, if you think about it. If you get a minus sign after an operand or a right parenthesis, it will signify the binary version, not the unary one. In our algorithm we shall have to track these states so that we can detect the context. What the code I've written does is to track three states: the next token *must* be an operand, the next token *could* be an operand, and the next token *cannot* be an operand. Using these states we can also detect errors as well (for example, once we've read an operand token, we can set the state to `Next token cannot be an operand` and thereby detect two operands in a row. So, maybe all is not lost.

Stay Awhile

It would seem that I can now show some code and discuss it; and ordinarily I would. However, I was reading a thread with messages from Chuck Jazdzewski, the lead developer and architect for Delphi, on a private newsgroup, and he mentioned a data structure called an ObStack for stacking or storing 'objects' of varying sizes like strings without incurring the overhead of separate allocations for the individual items (the message thread was about heap management for threads, but that's for another time).

I was writing this article at the time and was already a little put out at the thought of all the string allocations the parser required for the operand stack. Chuck's mention of ObStack got me searching the internet, and I found it in the source code for the GNU C/C++ compiler. Rather than describe the ObStack structure itself, I wrote a variant for Delphi that stacks Pascal length-byte strings: the `TaaStringStack`.

If you or I were to write a string stack, we'd probably do it in the same way: a linked list of pointers to strings on the heap. Something along the lines described in the *Algorithms Alfresco* column for February 1999, perhaps. Pushing a

```
type
  PChunkHeader = ^TChunkHeader;
  TChunkHeader = packed record
    chLimit : PChar;
    chPrev  : PChunkHeader;
  end;
  PStringNode = ^TStringNode;
  TStringNode = packed record
    snPrev   : PStringNode;
    snString : TaaString255;
  end;
function TaaStringStack.Pop : TaaString255;
var Temp : PChar;
begin
  {check for the obvious mistake}
  if (FCurString = nil) then
    raise Exception.Create(
      'TaaStringStack.Pop: the stack is empty');
  {return the current string}
  Result := PStringNode(FCurString)^.snString;
  {move current string pointer back, checking for
   switching chunks where we need to free chunk just left}
  if (FChunk + sizeof(TChunkHeader) = FCurString) then begin
    {we're leaving this chunk; set current string pointer}
    FCurString := PChar(PStringNode(FCurString)^.snPrev);
    {reset chunk address and dispose of the one just left}
    Temp := FChunk;
    FChunk := PChar(PChunkHeader(FChunk)^.chPrev);
    FreeMem(Temp, FChunkSize);
  end else begin
    {just move the current string pointer back}
    FCurString := PChar(PStringNode(FCurString)^.snPrev);
  end;
  dec(FCount);
end;
function TaaStringStack.Push(const aSt : TaaString255) :
  PaaString255;
var
  PrevNode      : PStringNode;
  NewCurString : PChar;
begin
  {save the current string node address}
  PrevNode := PStringNode(FCurString);
  {check for an empty stack}
  if (FCurString = nil) then begin
    if (FChunk = nil) then
      ssAddNewChunk;
  end else begin
    {advance the current string pointer}
    NewCurString := PChar(PrevNode) +
      sizeof(pointer) + length(PrevNode^.snString) +
      1 {the length byte};
    {align the new pointer}
    NewCurString :=
      pointer((longint(NewCurString) + 3) and $FFFFFFFC);
    {if there's not enough room for the new string, get a
     new chunk}
    if (PChunkHeader(FChunk)^.chLimit - NewCurString) <
      (sizeof(pointer) + length(aSt) + 1) then
      ssAddNewChunk
    {otherwise, position the current string pointer}
    else
      FCurString := NewCurString;
  end;
  {set up the new node}
  with PStringNode(FCurString)^ do begin
    snPrev := PrevNode;
    snString := aSt;
  end;
  {return address of the pushed string}
  Result := PaaString255(FCurString + sizeof(pointer));
  inc(FCount);
end;
```

➤ *Listing 1: Pushing and popping from a string stack.*

string would mean allocating a new node, allocating space on the heap for the string inside the node and then adding the node to the head of the stack. In the prior column I showed you how to allocate blocks of nodes from the heap, in order to save time and space. But what about the strings? How can we allocate 'blocks' of them? I would discount allocating a whole batch of 255 character strings and doling them out; the vast majority of RPN expressions we shall be creating would be much smaller than that.

This is where the ObStack comes in; at least, the variant I'll describe. The short string stack is an abstract data structure with four main methods: pushing a string, popping a string, finding if the stack is empty, calculating the number of items in the stack. The class we'll describe interfaces these standard methods, but implements them in a clever way.

What we shall do is this: allocate a large chunk of memory on the heap, say 4096 bytes. When we push a string onto the stack, internally we find the end of the last string pushed, and then copy the string being pushed immediately afterwards. Eventually we shall run out of space in this chunk, the string we're trying to copy is too large for the remaining space, and we shall allocate another chunk of memory and continue the same process. The chunks of memory will be chained together in a singly-linked list in the order we allocate them.

All very well, but what happens when we pop the strings? Under the scheme I've just described we'd be in a pretty pickle: we have no way of knowing how long previous strings are and hence how to go back through the stack. Even worse: what happens if the previous string is in another chunk? The obvious answer is that we need more structure to the chunks. The class I designed stores string nodes in the chunks. Each node consists of a pointer to the previous node, followed by a variable number of bytes storing the string.

```
TaaExpressionParser = class
  public
    constructor Create(const aExpr : string);
    destructor Destroy; override;
    property Expression : string
      read epGetExpression write epSetExpression;
    property RPNExpression : string
      read epGetRPNExpression;
    property Value : double
      read epGetValue;
    property Variable[const aName : string] : double
      read epGetVariable write epSetVariable;
  end;
```

➤ *Listing 2: Expression parser interface.*

The code is shown in Listing 1. When we push a string, the class works out the address of the next node in the chunk. For efficiency purposes, we make sure the node starts on a 4-byte boundary. The class then calculates the amount of space required (being 4 bytes for the pointer, a byte for the string length and then the actual characters in the string) and checks that enough room is left to store the string. If not, a new chunk is allocated, linked to the previous chunk, and the start of the chunk is the position for the new node. The pointer for the new node is set to the address of the previous string node, and the string is copied after the pointer (notice that the string itself starts on a 4-byte boundary). This new string is now at the top of the stack. For convenience, we return the address of the string we just pushed on the stack.

Popping is slightly more complicated than just following the pointer in the top string node. If we move back from one chunk into the previous chunk, we make a note of this and make sure that the chunk we've just emptied is freed.

Those of you who've been following the code can probably see that it's pretty nasty looking. To ease the calculation of addresses I've used `PChars` all over the place because Delphi only supports pointer arithmetic on `PChar` pointers. It can look pretty scary at times, what with all the typecasts going on, but the above description and the comments should help you follow the logic. Those of you who have been *really* following the code will see that I am also making the strings pushed on the stack null terminated for convenience.

What's It Gonna Be

The full expression parser requires two other stacks: an operator stack (which, in fact, can be a simple character stack) and a stack of double values. I hesitate to call the latter a 'double stack' since it conveys the wrong impression (two stacks side by side?), so I'll call it a float stack instead. We might as well write these two stacks as separate classes that we can reuse later if we so wish. I won't show the code here; it's all pretty simple and the units are on this month's disk.

After all that, I think it's time I showed some expression parsing code. Listing 2 shows the interface of an expression parser. The `RPNExpression` property returns the RPN version of an ordinary algebraic expression as a specially formatted string. The reason for the 'special' formatting is that the parser can parse expressions like 25+42 to give the very ambiguous 2542+ if we weren't very careful. The formatting used is that all atomic operands (like numbers) in the RPN expression are prefixed by a space, so that they can be easily separated when it's time to evaluate the expression. 25+42 would then come out as ' 25 42+'. Listing 3 reveals the higher-level code that generates the RPN expression. A couple of notes might be in order: `epFormRPNSubExpr` generates an RPN expression from the operator in hand and the operand stack, `epCheckBadParserState` just checks the parser state is valid (and raises an exception, if not) and `epGetPrecedence` is a simple lookup routine that returns the precedence level of the operator passed to it.

Son Of A Preacher Man

The expression parser class I'm providing has an extra bit of functionality: it allows the use of variables in the expression string. If we wish to evaluate an expression containing variables, we shall need to provide a way of informing the parser object the values for the various variables. That's simple enough (a method with two parameters, one a variable name

➤ *Listing 3: Creating the RPN expression.*

```
procedure TaaExpressionParser.epParseToRPN;
var
  ParserState : TaaExprParserState;
  TokenType   : TaaExprTokenType;
  Op          : char;
  StartPos    : PChar;
  PrecOp      : integer;
  PrecTop     : integer;
begin
  {if we've done this already, get out}
  if FParsed then Exit;
  {initialize the operator stack to have a left parenthesis;
   when we reach the end of the expression we'll be
   pretending it has a right parenthesis}
  FOpStack.Clear;
  FOpStack.Push('(');
  {initialise the operand stack}
  FStStack.Clear;
  {initialise the parser}
  FExpr := FOrigExpr;
  ParserState := psCouldBeOperand;
  {get the next token from the expression}
  TokenType := epGetNextToken(StartPos);
  {process all the tokens}
  while (TokenType <> ttEndOfExpr) do begin
    {what type of token are we trying to parse?}
    case TokenType of
      ttOperator :
        begin
          {it's an operator}
          Op := StartPos^;
          {if the operator is a left parenthesis, just push
           it onto the operator stack}
          if (Op = '(') then begin
            FOpStack.Push(Op);
            ParserState := psCouldBeOperand;
          end else begin
            epCheckBadParserState(ParserState,
              psMustBeOperand, StartPos);
            {if the operator is a right parenthesis, start
              popping off operators and operands and forming
              RPN subexpressions, until we reach a left
              parenthesis}
            if (Op = ')') then begin
              if FOpStack.IsEmpty then
                epRaiseBadExpressionError(StartPos);
              epFormRPNSubExpr(')', StartPos);
              ParserState := psCannotBeOperand;
            end
            {if the operator is a unary operator, then
             ignore a unary plus (it has no effect) and
             push a unary minus}
            else if (ParserState = psCouldBeOperand) then
              begin
                if (Op <> '+') and (Op <> '-') then
                  epRaiseBadExpressionError(StartPos);
                if (Op = '-') then
                  FOpStack.Push(UnaryMinus);
                ParserState := psMustBeOperand;
            end else begin
              {if we reach this point, the operator must be
               pushed onto the stack, however, we first need
               to check that we are not pushing it onto an
               operator of greater precedence}
              PrecOp := epGetPrecedence(Op);
              if FOpStack.IsEmpty then
                PrecTop := 0
              else
                PrecTop := epGetPrecedence(
                  FOpStack.Examine);
              if (PrecOp < PrecTop) then
                epFormRPNSubExpr(Op, StartPos);
              FOpStack.Push(Op);
              ParserState := psCouldBeOperand;
            end;
          end;
        end;
      ttNumOperand, ttVarOperand :
        begin
          {it's an operand}
          epCheckBadParserState(ParserState,
            psCannotBeOperand, StartPos);
          epPushNewOperand(StartPos);
          ParserState := psCannotBeOperand;
        end;
    end;
    {get the next token from the expression}
    TokenType := epGetNextToken(StartPos);
  end;
  {at the end we pretend that the expression was terminated
   with a right parenthesis and we can't be expecting an
   operand}
  epCheckBadParserState(ParserState, psMustBeOperand,
    StartPos);
  epFormRPNSubExpr(')', StartPos);
  {at this point, the operator stack should be empty and the
   operand stack should have one item: the RPN of the
   original expression}
  if (not FOpStack.IsEmpty) or (FStStack.Count <> 1) then
    epRaiseBadExpressionError(StartPos);
  FParsed := true;
end;
```

and the other a value!). The more difficult problem is internal: how to store this information.

The data structure we use must consist of a set of variable names and for each variable name there is associated a value of `double` type. It would be easy to use a `TStringList` and allocate a `double` variable on the heap for each variable, but that's a little inefficient. One alternative is to allocate chunks of `double` values (say 100 at a time) and store the address of a `double` variable with the string in the string list. This mimics the method we used with linked lists in February 1999's *Algorithms Alfresco* where we allocated chunks of nodes at a time, rather than singly. However, since the variable names we shall be using are all very short (obviously shorter than the expression in the first place) we'll reuse the string stack instead, in a data structure of our own devising (maybe an even better alternative would be to use a hash table...).

By the way, by arguing several possibilities like this I hope to show that there is never any single best way to perform a particular programming task. Sometimes, you will use one method and data structure, at other times another algorithm will suggest itself. Only by learning about data structures and algorithms and understanding their ramifications are you able to see the different possibilities; I hope that *Algorithms Alfresco* plays its part in this.

Anyway, enter the variable list class. We'll make use of Delphi's ability to declare properties that look like arrays and furthermore to accept a string as the index. To set the variable *x* you'd write:

```
MyVarList['x'] := 1.0;
```

and to read its value later:

```
Value := MyVarList['x'];
```

All very bizarre looking, but very, very handy.

Inside is where the interesting stuff happens, as usual. We'll declare an array type of nodes, each node will contain a pointer to a shortstring (the pointer will be provided by the string stack's `Push` method) and a `double` value. The variable list class will allocate one of these arrays internally. New variable names will be inserted into the array in sorted order; this makes finding a particular variable name very fast, we can use the normal binary search routine. So the write method for the property we described above will attempt to find the variable in the array. If it was found, the value in that element will be overwritten with the new value. If it wasn't found, then we push the variable name onto the string stack, which gives us the address of the string in the stack's internal data structure, and we insert a new element at the proper place in the array. Reading the value of a variable proceeds in the same fashion: find the name in the array; if found, return the value; if not, return zero (we could raise an exception for the latter case, but I consider it 'nicer' to be less aggressive and return zero, it's as if all possible variables exist and are pre-initialized to zero).

At this point we can start evaluating an expression: we have a way of setting values for various variables, we have a way of parsing the expression into the more efficient RPN form, we know (in theory) how to evaluate an RPN expression. The evaluator method is shown in Listing 4.

Nothing Has Been Proved

So, let's move onto my brother-in-law's homework. If you recall, we had to use all the digits 2, 3, 5, and 7, and any of the usual arithmetic operators to make all the numbers from 1 to 100 (or at least as many as we could possibly do). My approach to this problem was to generate RPN expressions by using combinations of the four allowable operands together with any of the standard operators. Since the normal arithmetic operators are all binary in nature (we shall ignore the possibility of using the unary minus), it turns out that the RPN expressions would all have three operators to go with the four operands.

The program has to generate all possible RPN expressions of the form *aabbbbc* where *a* stands for an operand (one of the four

➤ *Listing 4: Evaluating an RPN expression.*

```
function TaaExpressionParser.epGetValue : double;
var
  DblStack : TaaFloatStack;
  i        : integer;
  Operand1 : double;
  Operand2 : double;
  Expr     : string[255];
  OperandSt: string[255];
begin
  if not FParsed then
    epParseToRPN;
  {prepare a stack for doubles}
  DblStack := TaaFloatStack.Create;
  try
    {read through the RPN expression and evaluate it}
    Expr := FStStack.Examine;
    i := 0;
    while (i < length(Expr)) do begin
      inc(i);
      if (Expr[i] = ' ') then begin
        if Expr[i+1] in NumberSet then begin
          OperandSt := '';
          while Expr[i+1] in NumberSet do begin
            OperandSt := OperandSt + Expr[i+1];
            inc(i);
          end;
          DblStack.Push(StrToFloat(OperandSt));
        end else begin
          OperandSt := '';
          while Expr[i+1] in IdentifierSet do begin
            OperandSt := OperandSt + Expr[i+1];
            inc(i);
          end;
          DblStack.Push(FVarList.Value[OperandSt]);
        end
      end else begin
        if Expr[i] = UnaryMinus then
          DblStack.Push(-DblStack.Pop)
        else begin
          Operand2 := DblStack.Pop;
          Operand1 := DblStack.Pop;
          case Expr[i] of
            '+' : DblStack.Push(Operand1 + Operand2);
            '-' : DblStack.Push(Operand1 - Operand2);
            '*' : DblStack.Push(Operand1 * Operand2);
            '/' : DblStack.Push(Operand1 / Operand2);
            '^' : DblStack.Push(Power(Operand1, Operand2));
          end;{case}
        end;
      end;
    end;
    Result := DblStack.Pop;
  finally
    DblStack.Free;
  end;
end;
```

```
procedure AddOperators(aProcess : ThwProcessExpression;
  var aExpr : ThwExpression);
var
  i, i1, i2, i3 : integer;
  FirstBlank    : integer;
  SecondBlank   : integer;
begin
  {find first and second blanks; third is always at 7}
  FirstBlank := 0;
  SecondBlank := 0;
  for i := 3 to 6 do begin
    if (aExpr[i] = ' ') then
      if (FirstBlank = 0) then
        FirstBlank := i
      else begin
        SecondBlank := i;
        Break;
      end;
  end;
  {replace blanks with every combination of operators}
  for i1 := 1 to length(Operators) do begin
    aExpr[FirstBlank] := Operators[i1];
    for i2 := 1 to length(Operators) do begin
      aExpr[SecondBlank] := Operators[i2];
      for i3 := 1 to length(Operators) do begin
        aExpr[7] := Operators[i3];
        {process the completed RPN expression}
        aProcess(aExpr);
      end;
    end;
  end;
  {reset blanks: permutation/replacement logic requires it}
  aExpr[FirstBlank] := ' ';
  aExpr[SecondBlank] := ' ';
  aExpr[7] := ' ';
end;
procedure PermuteOperands(aProcess : ThwProcessExpression;
  var aExpr : ThwExpression; aInx : integer);
var
  i, j : integer;
  Ch   : char;
begin
  if (aInx = 6) then begin
    AddOperators(aProcess, aExpr);
  end else
    for i := aInx to 6 do begin
      Ch := aExpr[i];
      aExpr[i] := aExpr[aInx];
      aExpr[aInx] := Ch;
      PermuteOperands(aProcess, aExpr, succ(aInx));
      aExpr[aInx] := aExpr[i];
      aExpr[i] := Ch;
    end;
end;
procedure GenerateExpressions(aProcessExpr :
  ThwProcessExpression);
var
  Expr     : ThwExpression;
  i1, i2   : integer;
  Ch1, Ch2 : char;
  Operands : String6;
begin
  {preset expression string to operands plus three spaces}
  Expr := '2357   ';
  Operators := '+-*/';
  {generate the first token: an operand}
  for i1 := 1 to 4 do begin
    {swap characters 1 and i1}
    Ch1 := Expr[i1];
    Expr[i1] := Expr[1];
    Expr[1] := Ch1;
    {generate the second token: an operand}
    for i2 := 2 to 4 do begin
      {swap characters 2 and i2}
      Ch2 := Expr[i2];
      Expr[i2] := Expr[2];
      Expr[2] := Ch2;
      {permute tokens 3 thru 6}
      PermuteOperands(aProcessExpr, Expr, 3);
      {swap characters 2 and i2 back again}
      Expr[2] := Expr[i2];
      Expr[i2] := Ch2;
    end;
    {swap characters 1 and i1 back again}
    Expr[1] := Expr[i1];
    Expr[i1] := Ch1;
  end;
end;
```

➤ *Listing 5: Generating RPN expressions through permutations.*

allowable digits), *b* stands for either an operand or an operator, and *c* stands for an operator. If you think about how the RPN expression is evaluated, you'll see that this format restricts the RPN expression to a valid one: we must have at least two operands on the stack before we get to an operator (the *aa*) and we must finish up with an operator (the *c*), otherwise we'll end up with two or more operands on the stack at the end of the expression (which is invalid). Once we generate an RPN expression, we know how to evaluate it, and we just keep a running check on the numbers we've managed to generate so far.

Note that I was talking from the hip just then. The above description does *not* always produce a valid RPN expression. Nearly always, but not quite. A counter example is 23++57+, where we shall run out of operands on the stack for the second + operator when we try to evaluate it. However, adding extra checks to the permutation code to avoid this problem will only slow down the program. The reason is that this code is recursive and the checks would be evaluated many, many times. The recursive routine would also become much more complex to understand. Better, I think, to generate the invalid RPN expressions and then reject them at evaluation time.

What Have I Done To Deserve This?

How do we create the possible permutations of the form *aabbbbc*? If you haven't seen permutation code before, it's quite easy once you know how (as is everything!).

What we shall do is to permute the string '2357   ' (ie '2357' with 3 spaces following) with the assumption that (eventually) spaces will be converted into operators and the first two tokens (or characters) must be operands. We'll also assume that the seventh character must always be a space, so in fact we only have to do the permutation on the first six characters.

Right then, here we go: the first character of the expression. We must permute through all the possible digits here. We assume that the digits are in positions 1 through 4 of the string (we shall make sure that they are). What happens is an example of a divide and conquer technique; in other words, restating the algorithm as some simple operation together with the same or similar algorithm on a smaller set. We can then keep on reducing the scope of the algorithm step by step, until it's really easy or trivial to code.

That was the layman's description, now let's see what the algorithm actually does. Permute characters 2-6. Swap character 1 with character 2. Permute characters 2-6. Swap characters 1 and 2 back again. Swap characters 1 and 3. Permute characters 2-6. Swap characters 1 and 3 back again. Finally swap characters 1 and 4. Permute characters 2-6. So, without knowing anything about the process or what even happens when we permute characters 2-6 (all we really need to 'know' is that it somehow *works*), you can see that we've generated all of the permutations with a digit in position 1.

And what about the algorithm that permutes characters 2 to 6?

Well, we do the same thing, but reduce the problem (note we can't touch character 1 at this point). Permute characters 3 to 6. Swap characters 2 and 3. Permute characters 3-6. Swap characters 2 and 3 back again; swap characters 2 and 4. Permute characters 3-6. Notice we've reduced the problem again to a simpler one: that of permuting characters 3 to 6. Also we've ensured that character 2 is one of the digits.

Permuting characters 3 to 6 operates on a similar method of swapping characters, permuting the remainder, and then swapping the characters back again. It can be done by a recursive routine.

Finally we get to the lowest level of the permutation code when we try and permute the single character at position 6. This is a no-brainer (there's only one permutation!) and we can now easily replace the spaces in the expression string with every single combination of operator. See Listing 5.

Once we have properly permuted an RPN expression we can easily evaluate it. Indeed, in this case, with operands and operators always one character in size, the evaluation becomes much easier. The `HOMEWORK.DPR` program on the disk generates a table of expressions that evaluate to the numbers 1 to 100.

Goin' Back

Finally, I'll quickly outline how to proceed on my programming course project, differentiation of an algebraic expression. First, generate the RPN version of the expression. Next, read through the RPN expression and apply the standard differentiation rules, but cast in an RPN format. Obviously, *a′* is 0 for any constant *a*, and *x′* is 1. For an example of a differentiation rule: the differential of *fg+* is equal to *f′g′+* where *f* and *g* are functions in *x*. Likewise, (*fg**)′ is *f′g*fg′*+*, and so on in a similar vein for the other operators. The algorithm boils down to a recursive application of these and similar differentiation rules. So, as an example, differentiating *2x*1+* (ie *2x+1*) with these rules would result in (remember, read this using single digits) *0x*21*+0+* which could be simplified to 2 without too much problem. One day, when I have time, I may just finish that project from so long ago. But *not* in FORTRAN. In Delphi.

I hope you enjoyed this foray into an alternative parsing technique, and if you have a young relation who's been given an arithmetic homework puzzle you can at least dazzle him with your calculation abilities. Printout carefully hidden behind your back, of course.

Julian Bucknall is not Dusty, is not In Memphis, and knows just what to do with himself: write an algorithms book, hopefully at your bookshop by the new Millennium. He can be reached at julianb@ turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.

© Julian M Bucknall, 1999