NONPROCEDURAL PROGRAMMING

B. M. Leavenworth
Computer Sciences Department
IBM Thomas J. Watson Research Center
Yorktown Heights, New York

ABSTRACT Nonprocedural programming involves the suppression of unnnecessary detail from the statement of an algorithm. The conventional representation of an algorithm as a step by step sequential procedure often obscures the essential nature of the procedure. In many cases, algorithms are more transparent when stated recursively, combinatorially or nondeterministically. The paper discusses these three styles of programming and gives examples of their use. The elimination of certain low level features of traditional programming and their replacement by these and other techniques (associative referencing, aggregate operators and pattern matching) is advocated in order to raise the level of algorithm description.

Introduction

Nonprocedural programming has many of the goals of structured programming: constructing programs that are easier to understand, modify and debug. In addition, it involves the suppression of unnecessary detail from the statement of an algorithm. The conventional representation of an algorithm as a step by step sequential procedure often obscures the essential nature of the procedure. In many cases, algorithms are more transparent when stated recursively, combinatorially or nondeterministically.

We see the problem solving process as composed of three components:

(1) statement of the problem

(2) statement of the solution

(3) efficient implementation of the solution

We are mainly interested from a programming point of view in the second step. Step (3) can in principle be carried out by an optimizing compiler, whereas the the transformation from step (1) to step (2) is a problem in Artificial Intelligence.

In any case, the end user must always satisfy himself either that his statement of the problem (1) or solution (2) is correct. This paper is concerned with techniques of programming which, by removing certain low level details, make this task of verification easier.

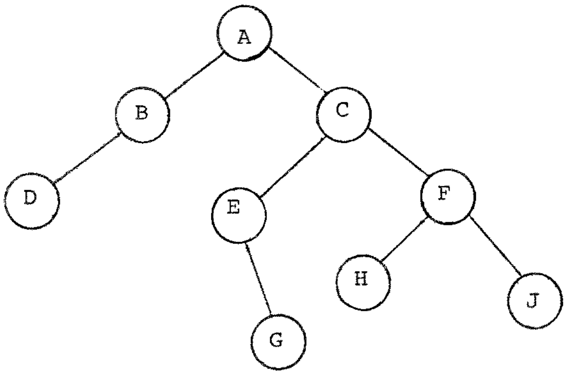
What is nonprocedural programming? There is no commonly accepted definition, but for purposes of this paper we will say that it involves specifying the outcome desired as a function of the inputs. A nonprocedural program is functional in the sense that it always produces the same output when presented with the same input; a nonprocedural program has no side effects. This condition can be guaranteed by eliminating assignment. For a survey of nonprocedural languages, see (Leavenworth and Sammet 1974).

Recursive Programming

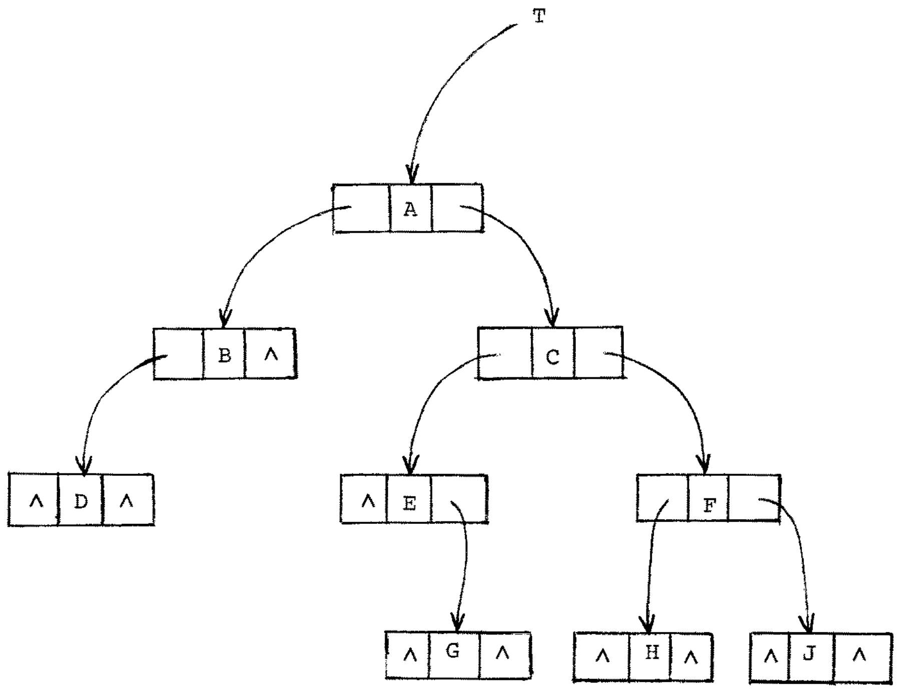
There are many algorithms that are easier to state recursively than iteratively. Examples abound in areas such as sorting, tree walking, parsing, etc.

Knuth's Chapter 2 on Information Structures (Knuth 1968) presents algorithms in a style which is suitable for efficient implementation.

As an example, we choose one of his algorithms for traversing binary trees. A binary tree is a finite set of nodes that either is empty, or consists of a root together with two binary trees. Knuth represents the tree



by the data structure:



The algorithm for "postorder" traversal can be stated simply:

```
Traverse the left subtree
Visit the root
Traverse the right subtree
```

The detailed version of this algorithm is given below in his iterative style.

Algorithm T. Let T be a pointer to a binary tree and A be an auxiliary stack.

```
T1.[Initialize.]Set stack A empty, and set the link variable P <- T.
T2.[P = ^?]If P = ^, go to step T4.
T3.[Stack <= P.] (Now P points to a nonempty binary tree which is to be traversed.) Set A <= P, i.e., push the value of P onto stack A. Then set P <- LEFT (P) and return to step T2.
T4. [P <= Stack.] If stack A is empty, the algorithm terminates; otherwise set P <= A.
T5.[Visit P.]"Visit" NODE (P). Then set P <- RIGHT (P) and return to step T2.
```

In this case, visit means accumulate the "value" of the root in a buffer which is printed when the algorithm terminates.

The above algorithm is reasonably close to a corresponding program in some high level language except that the stacking operations would be less clear in the program. It is representative of iterative algorithms with sequential updating of memory and transfer of control.

Since the treewalk is essentially a recursive procedure, we can describe the algorithm more naturally in a LISP-like functional language:

```
postorder x = if null (left x) then ()
                                else postorder (left x)
              || root x ||
              if null (right x) then ()
                                 else postorder (right x)
```

where '||' denotes an infix concatenation operator and 'left', 'root', and 'right' represent selectors of the three components of a node of the tree. The binary tree in this case can be constructed using either programmer-defined data types if such a facility exists (for example, SNOBOL4) or defining them by functional composition.

Recursive programming is supported by LISP and by the so called higher order languages (see next section).

Some further examples of recursive programming will be given in the next section.

Combinatory Programming

The idea of this type of programming is to manipulate and combine functions with the purpose of eliminating for the most part loops, conditionals and recursive calls (Burge 1972). By suppressing these "lower level" constructs, the programmer is freed from unnecessary detail and can exploit a powerful and concise style of programming.

In order to set the stage for examples of combinatory programming, consider the problem of adding up the elements of a list. The recursive algorithm is stated in English as follows:

To sum the elements of a list x, add the first element of x to the result of summing the remainder of x. This is translated into a recursive program as follows:

```
sum x = if null x then 0
                  else h x + sum (t x)
```

The boundary condition "if null x then 0" defines the identity element for addition and is always required for a recursive formulation. The above example is characteristic of recursive programming, but is "low level" in the sense that the recursive operation of the program involves data sequencing of the list.

The function just given is representative of a class of functions which can be defined using combinatory specification. Before explaining this technique, let us consider the slightly more complicated example of applying a given function f to each element of a list x. The recursive algorithm is:

To map a function f to each element of a list x, apply f to the first element of x and prefix this result to the result of mapping f to each element of the remainder of x.

The functional program is:

```
map f = λ x. if null x then ()
                          else f (h x) : map f (t x)
```

The interpretation is that the application of map to f produces a new function which when applied to a list x produces the desired result. This new function might be called an "f mapper". That is, it encapsulates (binds) the characteristics of f into the new function. Syntactically, however, it is more convenient to write the map function as follows:

```
map f x = if null x then ()
                    else f (h x) : map f (t x)
```

even though map (and every other function) is always applied to a single argument.

Now, the two preceeding dissimilar functions can be obtained as special cases of the following general list processing function:

```
list a g f x = if null x then a
                  else g ( f (h x)) (list a g f (t x))
```

Functions of this type which produce other functions as special cases will be called generators.

If the infix operators '+' and ':' are given the prefix formulations

```
     plus x y = x + y
     prefix x y = x : y
```

the previous functions can be defined in terms of 'list':

```
     sum = list 0 plus i
     map f = list () prefix f
```

where 'i' represents the identity function

```
    i x = x
```

The standard set operations have been defined by Burge (Burge 1968) using combinatory functions. We will give them here, since they demonstrate the flavor of combinatory programming. In what follows, sets will be represented by lists with no duplicate elements.

exists p = list false or $p^{\bullet}$

where 'or' is the logical function

or x y = <u>if</u> x <u>then</u> true <u>else</u> y

The 'exists' function applies the predicate p to each element of a list and returns true if at least one of the resulting values is true, and false otherwise.

filter p = list () i λ x. <u>if</u> p x <u>then</u> prefix x <u>else</u> i

The 'filter' function returns a subset of the argument set selected by the predicate p. Mathematically, the result is

$$\{x \in S \mid p(x)\}$$

where S is the argument set.

```
belongs l x = exists (equal x) l
```

where equal is the prefix formulation:

```
equal x y = x = y
```

The 'belongs' function is a predicate which returns true if x is an element of l, and false otherwise.

```
intsn = filter . belongs
```

where . is an infix representation of the prefix composition function

```
b f g x = f (g x)
```

Thus, f . g = b f g .

The 'intsn' function defines set intersection

```
diff x y = filter ( not . (belongs y)) x
```

where 'not' is the logical function

```
not x = if x then false else true
```

The 'diff' function defines set difference.

```
union x y = concat (diff x y) y
```

where 'concat' is defined by

```
concat x y = list y prefix i x
```

The 'union' function defines set union.

The type of combinatory functions described here are supported by and can be programmed in any of the "higher order" languages (Reynolds 1972) inspired by Landin (Landin 1966) such as PAL (Evans 1968), McG (Burge 1968), GEDANKEN (Reynolds 1970) and QUEST (Fenner et al 1972).

Nondeterministic Programming

The programming of a wide class of combinatorial problems is made easier by using certain operators introduced by Floyd (Floyd 1967). These consist of:

(1) a multiple valued choice function called choice (n) whose values are the integers from 1 to n

(2) a success function, and

(3) a failure function

The choice function allows a program to be conceptually executed in parallel with each path using one of the values of the argument. The success and failure functions label termination points of the computation. However, only those termination points labelled as success are considered to be computations of the algorithm.

Since context-free languages are recognized by nondeterministic pushdown automata, we will use these nondeterministic primitives in specifying a context-free parser.

We will modify the choice function slightly and allow the argument to be a list instead of an integer. Then each path will use one of the elements of the list.

The parsing algorithm to be programmed uses a top down method which will parse strings generated by any context-free grammar without left recursive rules. There is an input string and a prediction string which initially consists of the distinguished symbol S of the grammar. The leftmost symbol of the prediction string is tested for the following cases:

(1) If a Terminal symbol, it is compared with the input symbol under scan. If there is a match, both symbols are deleted, otherwise the failure function is invoked.

(2) If a Non-terminal symbol, it is replaced (using the choice function) by all the right hand rules defining it.

(3) If a Rule number, it is deleted from the prediction and added to the buffer.

A simple program to realize this algorithm will now be shown.

```
parse (input,pred,bufr) =
   if and (null pred,null input) then
          (print bufr; success) else
   if or (null pred,null input) then failure else
   if rule no (h pred) then
          parse (input,t pred,h pred:bufr) else
   if term (h pred) then if h input = (h pred)
                         then parse (t input,t pred,bufr)
                         else failure
   else let x = choice (gmap (h pred));
        parse (input,x || (t pred),bufr)
```

If both the prediction string and input string are empty, the buffer is printed (side effect !) and the parse is successful. After this test, if either the prediction string or input string is empty, the parse fails. If the top of the prediction is a rule number, it is added to the buffer. If the top of the prediction is a nonterminal, the choice function is called with a list of right hand rules as argument.'gmap' is a function (relation) that maps a left part (nonterminal) of a grammar to a list of alternatives (right hand rules). When the computation terminates, the buffer contains in reverse order the rules that were applied during the parse.

The above program, when defined in the environment of the grammar

gmap = {<S,([1aAS],[2a])>,<A,([3SbA],[4ba],[5SS])>},

representing the context-free grammar

S -> aAS | a

A -> SbA | ba | SS

and applied to the arguments input =[aabbaa] ,pred = S:() and bufr = () , produces the string 13242. 'gmap' is expressed as a binary relation where the range values are given as lists of character strings (nonterminals in upper case, terminals in lower case, and rule numbers denoted by integers).

Nondeterministic functions such as those described here have been added to FORTRAN (Cohen and Carton 1974). The approach followed is to transform programs written in the extended FORTRAN into standard (deterministic) FORTRAN following Floyd's work. Similar techniques can be applied to other high level languages.

Elimination of Low Level Detail

The programming techniques already introduced (recursive, combinatory, nondeterministic) do much to eliminate inessential detail in the programming process. Now we will briefly outline those low level features that can be eliminated and roughly their nonprocedural equivalents in the following form: low level feature => nonprocedural substitute. Some of the nonprocedural techniques have already been discussed. The others will appear in subsequent sections.

Explicit referencing and search => associative referencing

We would like to eliminate explicit access paths and referencing dependent on array subscripts, pointers and explicit searching.

Loops => associative referencing, aggregate operators, and combinatory programming

Elimination of loops raises the level of programming because it decreases the number of decisions the programmer has to make. We also include in this category most iterative and recursive constructions.

Explicit sequencing => recursive and combinatory programming

Explicit sequencing is intimately connected with procedural programming, side effects and the updating of memory. The presence of side effects increases greatly the opaqueness of programs and difficulty of verification.

Explicit control and pattern matching => nondeterministic programming and pattern matching

Pattern matching and nondeterministic control are treated together since they are related in many ways. The suppression of control flow is a step in the direction of nonprocedurality and serves to hide details which are not relevant to the problem solution.

Associative Referencing

We use the term associative referencing to refer to the accessing of data based on some intrinsic property of the data. This method of referencing allows the programmer to suppress implementation oriented details so that the decision of how to represents objects in the machine is left to the compiler.

The relation 'gmap' in the previous example represented a mapping from nonterminals to right hand rules in a grammar which did not commit the compiler to any particular representation or access paths. Earley (Earley 1974) has described higher level data structures (tuples, sequences, sets, relations) and operations on these structures which provide this type of freedom from access path dependence.

Associative referencing is usually described syntactically using the standard set notation:

$$\{x \in S \mid p(x)\}$$

That is, all the members of set S satisfying the property p(x). Underlying this syntax, however, is the application of a function such as 'filter' previously defined:

```
filter p S
```

<u>Aggregate Operators</u>

The set operators previously defined by combinatory functions are examples of aggregate operators. We will briefly discuss four types of aggregate operators which perform the following kinds of mappings:

(1) aggregate -> scalar

(2) aggregate -> aggregate

(3) aggregate X aggregate -> scalar

(4) aggregate X aggregate -> aggregate

An example of the first type is the reduction operator of APL which is exemplified by the 'sum' function defined earlier. The 'map' function is an example of type (2).

We will now define a function analogous to the 'list' function but which operates on two lists of equal length. It will then accommodate the two remaining types as special cases.

```
lists a g f x y = if null x then ()
                  else g (f (h x)(h y)) (lists a g f (t x)(t y))
```

An example of the third type is an inner product function defined as follows:

```
inner = lists 0 plus mult
```

where 'mult' is given by the prefix formulation

```
mult x y = x * y
```

Finally, a distribution function which applies the same operator on pairwise elements of two lists to produce a result list, a la APL, can be defined

```
dist f = lists () prefix f
```

It is well known that APL has excellent facilities for aggregate operations. However, the present approach is more powerful because <u>any</u> function can be the argument of a generator whereas the arguments allowed by APL are restricted to the built-in functions.

Pattern Matching

The string pattern matching facilities provided by SNOBOL4 (Griswold et al 1968) are representative of the type of operations we want in order to suppress low level detail. However, we would like pattern matching to be applicable to arbitrary data structures, not just strings.

The following highly recursive SNOBOL program which uses patterns and unevaluated expressions to recognize strings generated by the context-free grammar previously introduced, demonstrates the power of a generalized pattern matcher.

```
  &ANCHOR = 1
  A = *S 'B' *A | 'BA' | *S *S
  S = 'A' A *S | 'A'
  INPUT S RPOS(0)
END
```

In the above program, the grammar is represented by the pattern variable S, the infix symbol '|' represents the operation of alternation, and the unary operator '*' postpones evaluation of its operand. The first statement specifies "anchored mode", which means that the pattern must match the input string starting at the first character. Finally, the function call 'RPOS(0)' is a pattern that succeeds only if the entire input string has been scanned.

Unfortunately, the programmer can't use this mechanism to produce a parse because there is no way to distinguish between successes or failures of alternative paths. In addition to the information that the pattern matched or did not match the input string, it would be useful if SNOBOL produced a derivation tree as the value of the match which would indicate exactly which rules were used to match the given string.

An approach which incorporates a SNOBOL-like pattern matching facility into a higher order programming language called QUEST has been described by Tennant (Tennant 1973).

This approach allows the type of translation discussed above and hence is more powerful than SNOBOL.

Since space precludes an adequate discussion of pattern matching techniques, a discussion of their application in various artificial intelligence languages can be found in (Bobrow and Raphael 1973).

Summary

We have discussed in some detail three styles of programming subsumed by the notion of nonprocedural programming. These techniques have been applicable to raising the level of algorithm description. We have also advocated the elimination of certain low level features conventionally used and their replacement by these and other techniques such as associative referencing, aggregate operators and pattern matching. Finally, we have suggested some programming languages and extensions which support this type of programming.

REFERENCES

D.G. Bobrow and B. Raphael, "New Programming Languages for AI Research", Tutorial presented at 3rd IJCAI, Stanford, California, August, 1974.

W.H. Burge, "McG - A Functional Programming System", Report RC 2189, IBM Research Division, Yorktown Heights, N.Y.August 1968.

W.H. Burge, "Combinatory Programming and Combinatorial Analysis", IBM Journal of Research and Development, Vol. 16, No. 5 (Sept. 1972).

J. Cohen and E. Carton, "Non-deterministic FORTRAN", Computer Journal, Vol. 17, No. 1, (May 1974).

J. Earley, "Relational Level Data Structures for Programming Languages", Acta Informatica Vol. 2 Fasc. 4 1973.

A. Evans, "PAL - A Language designed for teaching Programming Linguistics", Proceedings ACM 23rd National Conference, 1968.

T.I. Fenner, M.A. Jenkins and R.D. Tennent, "QUEST : The Design of a Very High Level Pedagogic Programming Language", SIGPLAN Notices, Vol. 8, No. 2(Feb. 1973).

R.W. Floyd, "Nondeterministic Algorithms", JACM Vol. 14 (Oct. 1967).

R.E. Griswold, J.F. Poage and I.P. Polonsky, The SNOBOL4 Programming Language, Prentice-Hall, Englewood Cliffs, New Jersey, 1968.

D.E. Knuth, "Fundamental Algorithms", Vol. 1, The Art of Computer Programming, Addison-Wesley, Reading, Mass.,1968.

P.J. Landin, "The Next 700 Programming Languages", CACM Vol. 9, No. 3 (March 1966).

B.M. Leavenworth and J.E. Sammet, "An Overview of Nonprocedural Languages", Proceedings Symposium on Very High Level Languages, SIGPLAN Notices Vol. 9, No. 3 (April 1974).

J.C. Reynolds, "GEDANKEN : A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept", CACM Vol. 13, No. 5

J.C. Reynolds, "Definitional Interpreters for Higher-Order Programming Languages", Proceedings 27th ACM National Conference, 1972.

R.D. Tennent, "Mathematical Semantics and Design of Programming Languages", PhD. Thesis, University of Toronto, 1973.