literate programming

Moderated by
Christopher J. Van Wyk

EXPANDING GENERALIZED REGULAR EXPRESSIONS

Moderator's Introduction to Column 3

The mail prompted by Column 2 (*Communications*, December 1987, p. 1000) showed a broad array of opinions about what makes a good solution to a problem. Some readers found Jackson's program too closely tied to the representation of the input data, while others thought that it did not take enough advantage of those same details. I was heartened to hear from readers who preferred my pseudo-code solution, even though it was sketchy and incomplete.

Some correspondence suggests that I need to correct a few misimpressions. First, the solutions that appear in this column are in no way certified to be "the best" or "perfect"; they are merely the product of their authors' best attempts to produce literate programs. Second, I am not wed to the idea that what makes a program literate is the interleaving of code and nicely typeset comments; I would be delighted to hear from readers with other suggestions about how programs can be written literately—see address information at the end of this column.

This column's problem was posed by Mark Kahrs of Rutgers University.

> Here is a sample string together with the sequence of strings that it abbreviates: sample string: `a⟨8-10⟩[d-b]`; sequence: `a08d, a08c, a08b, a09d, a09c, a09b, a10d, a10c, a10b`. The sequence is derived from a string according to these rules:
>
> 1) Constant components (like a) always appear.
>
> 2) Numeric components (like ⟨1-3⟩) generate a sequence in the natural order (increasing or decreasing as appropriate) from the first component to the second component; the number is padded by zeroes on the left so that all elements of the expansion contain the same number of digits.
>
> 3) Alphabetic components (like `[d-b]`) generate a sequence in the natural order (increasing or decreasing as appropriate) from the first component to the second component.
>
> The sample string can contain any number of components.
>
> Write a function that takes a string and returns a list of the expanded strings. When I ask for a *function* I mean that I'd like the solution to be self-contained so that it can be plugged into a program without much fuss.

This problem arose in the implementation of a language for describing circuits. The sample string names a collection of circuit components in an order relevant to the processing of the circuit, and each component needs to have some storage reserved for it.

Kahrs mentioned that people had solved this problem in different ways. Some viewed the sample string as a generalized regular expression and wrote a program based on finite automata that produces all strings that match the regular expression. Others wrote a recursive program to solve the problem.

Eric Hamilton works in a group at Data General that has been developing software using tools that interleave code and design information. He volunteered to solve this problem using his group's tools.

Don Colner of Polaris reviewed Hamilton's program. The first review he sent contained several pieces of rewritten code. As natural as it may be to comment on a program by comparing and contrasting code fragments. I thought that rewriting the program went a bit further than a review ought to go. Colner graciously provided the revised review that appears here.

THE PROBLEM

The specification of how sequences are derived from a sample string is informal but clear. As with most informal specifications, it is not complete. I assume decimal arithmetic and interpret "padded by zeroes" to mean "padded by the minimal number of zeroes" because to do otherwise would be perverse. I restrict alphabetic components to the lower-case letters and constant components to alphanumeric characters and blank; because these are arbitrary choices I should be prepared to change them arbitrarily.

With these clarifications out of the way, we can consider the interface. Here, Van Wyk's problem statement gives us more freedom (or less guidance).

I have chosen to implement the solution as a function in C to run on Unix because these are plausible choices and because I have convenient access to a Unix system. Once this decision is made, the "without much fuss" requirement suggests that the input string and the returned strings be represented in the traditional C style, as a pointer to an array of characters terminated by the null character. Likewise, the returned list will be represented as a pointer to an array of such pointers, terminated by the null pointer. If the input string cannot be expanded for any reason the null pointer will be returned.

Since our function returns a pointer (a pointer to pointers, no less), we must consider the persistence of the denoted storage. We choose to guarantee that the pointer returned from one call to the expansion function will remain valid until the next call. We make this choice because it is easily implemented by reusing the same global storage on every invocation, relieves the caller of the burden of storage deallocation, prevents memory leaks, and simplifies the interface. This decision has been strongly influenced by the choice of C as the implementation language.

The solution presented here assumes that the input string is syntactically correct. This is because I am lazy and because the code to detect and recover from invalid input adds more clutter than interest. The argument string may already have been validated, or it may have been produced in a way which ensures its validity, so this assumption is not unreasonable. Because the caller has no way to ensure that the expansions will fit in the storage allocated for them, it is unreasonable not to check for overflow; the null pointer will be returned in this case.

Plan of Attack

We are going to attack the problem by generating expansions of the input string and saving them away; when all expansions have been generated, we will return the list of saved expansions. At this point, our subroutine looks like:

```
char ** expand_string(input_string)
char * input_string;
{
  /* Initialize list of expansions to
     empty */
  /* Generate expansions, adding them to
     list as they are found */
  /* Return list of expansions */
}
```

This code is little more than a restatement of the problem, yet it suggests a decomposition of the problem. We actually have two sub-problems. One is the generation of expansions from the input string, and the other is the management of a list (more precisely, a set) of strings. The second problem has been solved before, many times, and is pretty well under control, so we will start with the first. In solving it, we will find it necessary to decompose it into yet smaller sub-sub-problems, and so forth, until we arrive at problems which can be solved with C language primitives.

Generating Expansions

An unexpanded input string can be viewed as a compact description of a tree. Each expansion corresponds to the path from the root node to a particular leaf node. For example, ⟨8–10⟩[d–c] represents the tree shown in the accompanying diagram.

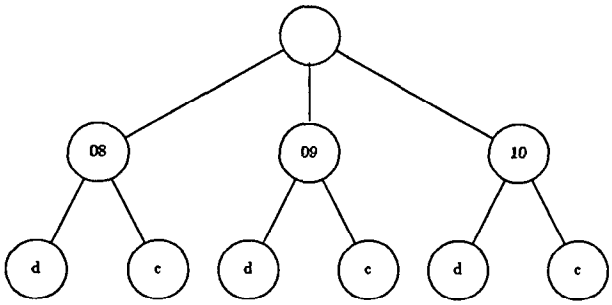

FIGURE 1

We can generate an expansion by walking the tree from the root to a leaf node and concatenating the labels of each node encountered in the walk. We can generate all expansions by performing such a walk to every leaf node in the tree.

Recursion is the natural way to exhaustively walk a tree, so the following program structure comes to mind:

```
recursively_expand
STRING input_string;
STRING generated_prefix;
{
  STRING s, unprocessed_components,
      current_component;
  if(input_string == NULL)
  {
    generated_prefix is an expansion;
    return OK;
  }
  else
  {
```

```
    current_component = first component
        of input_string;
    unprocessed_components = the rest of
        input_string;
    for s over all expansions of
        current_component repeat
    {
      status = recursively_expand(
                concatenate(
                   generated_prefix, s),
                unprocessed_components);
      if(status != OK)
      {
        return status;
      }
    }
  return OK;
  }
}
```

generated_prefix is the concatenation of the names of all nodes between the root and the current node. s is the name of the current node, and unprocessed_components is the unexpanded string describing the as yet unprocessed subtree rooted at the current node. The computation is started by calling recursively_expand with input_string equal to the string to expand and generated_prefix equal to the null string. Either the value OK or an appropriate error indicator will be returned.

The algorithm above is presented in psuedo-code because it depends on operations such as concatenation, "the rest of," and string assignment, which are not part of C. The next step must surely be to develop these abstract operations in terms of actual C primitives.

Managing the Strings

A careful examination of recursively_expand shows that it does only two things to strings. Characters are removed from the beginning of input_string and other characters are added at the end of generated_prefix. This behavior is unsurprising; we are just walking a path, generated_prefix is a list of steps already taken, and input_string specifies the steps yet to be taken. This behavior can also be coded in C quite easily. Incrementing a pointer to a string represented as a character array removes characters from the head of the string. Likewise, incrementing a pointer to the tail of such a string adds characters at the end. Thus, the string manipulations of recursively_expand can be expressed in terms of simple pointer manipulations.

Expanding a Single Component

The pseudo-code for recursively_expand blithely iterates s across all expansions of current_component. Unfortunately, this notion is harder to code than to pseudo-code. The components [a–c], [c–a], ⟨8–9⟩, ⟨9–8⟩, ⟨8–10⟩, and ⟨10–8⟩ all expand in slightly different ways. The code for handling each combination of counting up, counting down, padding with zeroes, not padding with zeroes, and so forth, threatens to be big, buggy, ugly, and uninteresting. This problem is inherent in the representation of each component as a character string; we would much rather speak in terms of things like:

```
current_component.initial_value
current_component.final_value
current_component.format_specification
```

And if we want to speak of such things, why not do so? We introduce a new data structure:

```
typedef struct
{
  unsigned  initial_value;
  unsigned  final_value;
  int       increment;
  char *    format_specification;
  unsigned  formatted_width;
  unsigned  component_width;
  component_descriptor_type;
}
```

to describe a component. The initial_value, final_value, and increment fields specify the range of values taken on by the component in question. The format_specification tells how to convert these values into a string suitable for use in an expansion. The formatted_width and component_width fields are the size of the component in its expanded and unexpanded forms.

This particular representation of a component works only if we assume that successive expansions of a component can be represented as consecutive integers; it will not work, for example, if characters were to be stored in EBCDIC instead of ASCII. We can relax this assumption by adding a field that specifies the way in which the initial_value and final_value fields are to be interpreted and iterated over.

Now we can write a function which inspects a component and returns the equivalent component descriptor:

```
component_descriptor_type
scan_first_component(input_string)
char input_string[ ];
{
  if(is_a_constant_component(
        input_string[0]))
  {
    return scan_constant_component(
        input_string);
  }
  else if(input_string[0] == '[')
  {
    return scan_character_component(
        input_string);
  }
  else if(input_string[0] == '<')
  {
```

```
    return scan_numeric_component(
        input_string);
  }
}
```

As a matter of convenience, a separate function is used to interpret each type of component, and `scan_first_component` simply determines the type of the first component and then invokes the function appropriate for that type. Thus, most of the useful work is accomplished in the functions `scan_⟨type⟩_component`:

```
component_descriptor_type
scan_constant_component(input_string)
char input_string[ ];
{
  component_descriptor_type
    return_value;
  return_value.initial_value =
    input_string[0];
  return_value.final_value =
    input_string[0];
  return_value.increment = 0;
  return_value.format_specification =
    "%c";
  return_value.component_width = 1;
  return_value.formatted_width = 1;
  return return_value;
}
```

A constant component is one character wide, as is its expansion, so both the `component_width` and the `formatted_width` fields are set to one. The rather cryptic `%c` is a format specification directing that a value be formatted as a single ASCII character. This and the following format specifications are processed by various library functions and are familiar to every C programmer.

```
component_descriptor_type
scan_character_component(input_string)
char input_string[ ];
{
  component_descriptor_type
    return_value;
  return_value.initial_value =
    input_string[1];
  return_value.final_value =
    input_string[3];
  if(input_string[1] <= input_string[3])
  {
    return_value.increment = 1;
  }
  else
  {
    return_value.increment = -1;
  }
  return_value.format_specification =
                                    "%c";
  return_value.component_width = 5;
  return_value.formatted_width = 1;
  return return_value;
}
```

All character components are five characters wide, but their expansion still occupies only one character, so the `component_width` and `formatted_width` fields are set accordingly. The handling of the case in which the `initial_value` and `final_value` fields are equal is arbitrary.

This implementation of `scan_character_component` assumes that the character collating sequence is arithmetical; this assumption is valid for the ASCII character set. It is easy enough to change this assumption, but the representation of the `component_descriptor_type` would have to be changed as well.

```
component_descriptor_type
scan_numeric_component(input_string)
char input_string[ ];
{
  int initial_width = 0;
  int final_width = 0;
  component_descriptor_type
    return_value;
  input_string++;
        /* Skip opening angle bracket */
  return_value.initial_value =
      atoi(input_string);
  while(*input_string != '>')
  {
    input_string++;
    initial_width++;
  }
  input_string++;
                      /* Skip the dash */
  return_value.final_value =
      atoi(input_string);
  while(*input_string != '>')
  {
    input_string++;
    final_width++;
  }
  if(return_value.initial_value <=
      return_value.final_value)
  {
    return_value.increment = 1;
  }
  else
  {
    return_value.increment = -1;
  }
  return_value.component_width =
      initial_width + final_width + 3;
  return_value.formatted_width =
      maximum(initial_width,
              final_width);
```

```
  if(return_value.formatted_width == 1)
  {
    return_value.format_specification =
      "%d";
  }
  else if(return_value.formatted_width
      == 2)
  {
    return_value.format_specification =
      "%02d";
  }
  return return_value;
}
```

The C library function `atoi` converts a string of ASCII digits terminated by any non-numeric character to the corresponding integer value. The two cases of the format specification provide the required zero padding; this code restricts numeric components to a width of two digits.

Generating Expansions, Again

At this point, the hierarchical decomposition of the string expansion problem is complete, and we can cast `recursively_expand` in C. As before, an initial call is required to begin the computation.

```
recursively_expand(input_string,
    generated_prefix_end)
char * input_string;
char * generated_prefix_end;
{
  char * unprocessed_components;
  component_descriptor_type
      current_component;
  int s;
  int status;
  BOOLEAN last_time;
  if(*input_string == '\0')
  {
    *generated_prefix_end = '\0';
          /* End-of-string indicator */
    status = string_is_an_expansion(
        generated_prefix);
    return status;
  }
  else
  {
    current_component =
        scan_first_component(
            input_string);
    unprocessed_components =
        input_string +
        current_component.
        component_width;
    s = current_component.
        initial_value;
    last_time = FALSE;
    while(!last_time)
    {
      if(s ==
        current_component.
        final_value)
      {
        last_time = TRUE;
      }
      sprintf(generated_prefix_end,
          current_component.
          format_specification, s);
      status = recursively_expand(
          unprocessed_components,
          generated_prefix_end +
          current_component.
          formatted_width);
      if(status != OK)
      {
        return status;
      }
      s += current_component.increment;
    }
    return OK;
  }
}
```

We have followed the pseudo-code closely, yet the semantic gap between it and this implementation is striking. Pointer arithmetic has replaced the string manipulations of the pseudo-code. Furthermore, the variable `generated_prefix` has degenerated into a fixed character array and a pointer into the array. The termination condition in the loop is expressed as "the previous iteration was the last"; this is because `s` may approach its final value from above or below, so that equality is the only useful test.

Storage Management

At this point, we have completely solved one of our two original subproblems. The other one, managing the list of generated expansions, is much easier. The routine `string_is_an_expansion` is called from `recursively_expand`; it accepts a string that is known to be an expansion and adds it to the list of expansions already generated. The pseudo-code for `expand_string` initializes the list to empty; the principle of information hiding demands that this be done with a subroutine as well.

```
char * string_list[
    NUMBER_OF_EXPANSIONS_LIMIT+1]
   /* The +1 allows for the endmarker */
unsigned number_of_expansions;
#define TOTAL_CHARACTER_LIMIT
    NUMBER_OF_EXPANSIONS_LIMIT *
    (EXPANSION_LENGTH_LIMIT+1)
char
character_pool[TOTAL_CHARACTER_LIMIT];
char * character_pool_first_free;
initialize_list_to_empty( )
```

```
{
  number_of_expansions = 0;
  character_pool_first_free =
      character_pool;
  string_list[number_of_expansions] =
      NULL;
  return;
}
string_is_an_expansion(string)
char * string;
{
  if(number_of_expansions >=
      NUMBER_OF_EXPANSIONS_LIMIT)
  {
    return STATUS_TOO_MANY_EXPANSIONS;
  }
  string_list[number_of_expansions] =
      character_pool_first_free;
  number_of_expansions += 1;
  string_list[number_of_expansions] =
      NULL;
  while(* character_pool_first_free++ =
      *string++)
    ;
  return OK;
}
```

Expansions are saved by consecutively copying them into the `character_pool` array. Pointers to each expansion are collected in the `string_list` array, and a pointer to this array will become the final return value.

Putting the Pieces Together

With the storage manager and `recursively_expand` written, the final implementation of the `expand_string` function which solves Van Wyk's problem is anticlimactic.

```
char
  generated_prefix[
    EXPANSION_LENGTH_LIMIT+1];
/* The +1 is for the end
                  of string indicator */
char ** expand_string(input_string)
char * input_string;
{
  int status;
  initialize_list_to_empty( );
  status = recursively_expand(
      input_string, generated_prefix);
  if(status == OK)
  {
    return string_list;
  }
  else
  {
    return(char **) NULL;
  }
}
```

Performance

There are many reasons to expect this implementation to be a real dog. It does an amount of work that grows exponentially with the number of components in the input string, and it calls `sprintf` an exponential number of times. Not only that, but it processes the same piece of the input string over and over; the number of calls to `scan_first_component` is exponential in the number of components that are there to be scanned. Expanding a constant component is little more than copying a byte, yet `recursively_expand` uses two procedure calls and an entire level of recursion to accomplish this task. Thus, it was with some trepidation that I approached performance measurement.

The size of the output generated by `expand_string` grows very rapidly with the number of components in the input string. A measurement of the time required to build the expansion of `[a-j] [a-j] [a-j] [a-j] [a-j] [a-j]` will tell us much more about the underlying virtual memory system than it will about the performance of `expand_string`. Thus, I replaced the call to `string_is_an_expansion` in `recursively_expand` with a call to the C library function `printf`, which simply prints its argument. I then ran `expand_string` on the input strings `[a-j] [a-j] [a-j] [a-j]` and `[a-j] [a-j] [a-j] [a-j] [a-j]`; these generated 10,000 and 100,000 bytes of output, respectively. I timed both while directing the output to the null device, and discovered that `expand_string` generated expansions at a rate of 2,500 characters per second. To put that figure in perspective, I also wrote a program that simply printed characters to standard output:

```
main(argc, argv)
int argc;
char *argv[ ];
{
  int i;
  i = atoi(argv[1]);
  while(i-)
    printf("%c", 'a');
}
```

This program proved capable of generating 7,500 characters per second. From this, I conclude that even the naive implementation of `expand_string` presented here is not unreasonably expensive.

Of course, there are things that can be done to improve its performance. Constant components are a special case; they can be copied from `input_string` to `generated_prefix_end` by adding the following code at the beginning of `recursively_expand`:

```
while(
  is_a_constant_component(
  *input_string))
{
  *generated_prefix_end++ =
  *input_string++;
}
```

Another, better, idea is to reduce the number of calls to `scan_first_component`. This routine can be invoked repeatedly on the initial input string to convert it into a string of component descriptors; the string of component descriptors can then be processed by the same recursive expansion already used. This is really just an application of the well-known technique of moving invariant computations out of a loop, or in this case, out of a recursion.

However, we must remember that the amount of output generated by this program is exponential in the size of the input. No amount of performance tuning can change the fact that this problem cannot be solved except in exponential time and space. It does not matter how small input strings are handled, and large input strings are hopeless; every ⟨0–9⟩ will burn another order of magnitude. Thus, it may be a mistake to put much effort into tuning `expand_string`. Instead we should concentrate on the application within which it is embedded.

Interface Improvements

Dean Herington and Andy Huber, also at Data General, independently suggested that the problem never should have called for returning a list in the first place. Herington suggested the interface to `expand_string` should include a user-supplied function which will be invoked whenever an expansion is generated. This function will process the expansion in accordance with the user's requirements; the processing might consist of storing it in a convenient, user-allocated, location. Huber suggested that the most useful interface is a generator function which, presented with a string and one of its expansions, returns the next expansion. Both approaches completely separate the problem of storage management from the problem of generating expansions. Huber's strategy is particularly good if the user intends to touch each expansion only once; the exponential storage requirement is eliminated.

I chose not to incorporate either because they do excessive violence to Van Wyk's problem statement; neither strikes me as returning a list "without much fuss." However, they are cleaner and more useful in a variety of situations. It is quite possible that either solution would have met the actual, as opposed to the stated, requirements of the problem. The moral of this is that some problems are best solved by negotiation with the user community.

Conclusion

This is a messy problem. The solution requires well over 100 lines of C code, yet can be stated in 20 lines of pseudo-code. The strategy behind the solution is obvious from the first glance at the tree structure derived from an input string; the same strategy all but disappears when the pseudo-code is fleshed out into real C. A truly discouraging amount of work is required to realize the intuitively obvious notion of iterating across all expansions of a given component. A gap between idea and implementation is, I think, typical in programming work. In this case, the gap is larger than usual.

It is interesting to consider how this problem might be solved in other programming languages. Fortran? I wouldn't consider it. Assembler? A horrible idea, but at least recursion would be available. C? Ten hours. I know, because I just did it. Ada? The semantic gap is much narrower. Lisp? An hour or so at the most. Snobol? Less than that. Those two languages have the right string and list manipulation primitives for the job. Unfortunately, we seldom have the luxury of fitting an implementation language to a problem. Van Wyk mentions that the solution to his problem is to be embedded within "the implementation of a language for describing circuits"; I doubt that this project will be coded in Snobol. C is a plausible choice, and closing the gap between the problem statement and a working C implementation is an interesting problem.

Eric Hamilton
Data General Corporation
62 T. W. Alexander Drive
Research Triangle Park, NC 27709

REVIEW

Eric Hamilton presents an elegant algorithm for creating the strings required by Van Wyk's problem. He structures the solution in terms of program modules that perform clearly separate, well-defined functions.

While this code provides an adequate solution to the algorithmic requirements of the problem, it fails to meet the problem statement's requirement that ". . . an argument string can be of any length." In addition, it fails to meet the environmental requirement, "that it can be plugged into a program without much fuss." In particular, this code presents the following problems:

1. Buffer overflow can cause unpredictable behavior.
2. Unanticipated side effects can result from the choice of extern rather than static external variables and functions.
3. The code is not reentrant.

Examples of Craftsmanship

The decision to return a null-terminated list of string pointers is conventional and sound. The choice of a recursive algorithm is natural, and the code is well-crafted.

Modularity

Eric Hamilton's functions are well designed modules. That is, each function performs a relatively simple action that can be understood without understanding the other functions. And, with a few exceptions, no module operates in another module's space. For example, each of the `scan_TYPE_component( )` functions builds a `component_descriptor_type` structure in its own local data area and the parent function copies the structure into its local data area on return. The exceptions to this policy are weak points in the design.

One of the best examples of craftsmanship in this

code is the isolation of the "big, buggy, ugly and uninteresting" details of dealing with three different types of input tokens. Each type of token is analyzed by a different function:

```
scan_constant_component( )
scan_character_component( )
scan_numeric_component( )
```

Each of these functions returns a structure that contains a generalized description of the substrings that must be concatenated onto the output string. The principal function, `recursively_expand( )`, is able to treat each token in the same way using these generalized descriptions.

Modularity Violated

Eric Hamilton tries to ignore some of the "ugly" details by limiting numeric components to a width of two digits. His code is:

```
if(return_value.formatted_width == 1)
{
  return_value.format_specification =
      "%d";
}
if(return_value.formatted.width == 2)
{
  return_value.format_specification =
      "%02d";
}
```

The data structure used here violates a principle of modularity. Successive invocations to this function can return pointers to the SAME STRING. Suppose the parent function decides to alter the format specification of decreasing numeric sequences:

```
for(k = 0; k < 3; input_string +=
    component[k++].component_width)
    {
    component[k] = scan_first_component(
                          input_string);
    if(component[k].
         format_specification[2] == 2 &&
       component[k].increment < 0
       )component[k].
          format_specification[2] = '3';
    }
```

If the input specification is "⟨9−10⟩⟨10−9⟩⟨9−10⟩", on return from the first invocation of `scan_first_component( )`, `component[0].format_specification` points to "`%2d`", but after the second invocation it points to "`%3d`". It appears that the programmer's intention is to produce an initial output string that looks like, "0901009" but the program will in fact produce "009010009". By returning a pointer to a string, this design permits the parent function to alter the behavior of the child. While this violation of the principle of modularity does not produce any problems in this program, it is a trap for some future maintenance programmer.

Restoring Modularity

As an alternative, I propose changing `format_specification` to an array of characters in `component_descriptor_type` and replacing Hamilton's code which builds `format_specification` to:

```
sprintf(
  return_value.format_specification,
  "%%0%ud", return_value.
  formatted_width);
```

The alternative code removes the restriction on the width of numeric components: it is shorter, it is less ugly, and it is safer than the original.

Buffer Overflow

The program checks for too many strings in the `string_list` array at the beginning of `string_is_an_expansion( )`, but it does not check for overflow of the `character_pool` or `generated_prefix` buffers.

`character_pool`

The `character_pool` buffer is declared as an external variable, and each new string added to the `string_list` is concatenated onto `character_pool` in `string_is_an_expansion( )`. This function should include a one-line test for buffer overflow.

`generated_prefix`

The `generated_prefix` buffer is declared as an external variable, and its starting address is passed to `recursively_expand( )` by `expand_string( )` to use as a working buffer to build the output strings. Each recursive instance of `recursively_expand( )` concatenates another substring onto `generated_prefix` until the end of the input specification is encountered, at which point `recursively_expand( )` invokes `string_is_an_expansion( )` to concatenate `generated_prefix` onto `character_pool`; `recursively_expand( )` should include a one-line test for buffer overflow.

One Test or Two

The `generated_prefix` buffer overflow test described in the previous section makes the `character_pool` buffer overflow test in `string_is_an_expansion( )` redundant. Why not eliminate the `character_pool` buffer overflow test and save a few processing cycles?

Eliminating the redundant buffer overflow test creates a new dependency between `string_is_an_expansion( )` and `recursively_expand( )`. Suppose a maintenance programmer later decides to build up the output string in a file rather than in a memory array. The length of `generated_prefix` will then be limited by the available disk file space rather than by `EXPANSION_LENGTH_LIMIT`. The maintenance programmer argues (incorrectly) that it

has been possible to eliminate a constraint and thereby simplify the user interface. The programmer will inadvertently introduce a bug into `string_is_an_expansion( )` as the `EXPANSION_LENGTH_LIMIT` constraint is eliminated.

This discussion of a maintenance programmer eliminating constraints is not academic. Subsequent sections of this review discuss the issue and eliminate the `EXPANSION_LENGTH_LIMIT` constraint.

I would argue for keeping the redundant buffer overflow check in `string_is_an_expansion( )` on the grounds that it increases the independence of the modules and therefore reduces the maintenance cost of the program. As rules of thumb, I figure that maintenance costs are ten times programming costs and that processing costs are zero. Adding an extra line of code and a few processing cycles at run time to reduce maintenance costs usually produces a very high benefit-cost ratio.

Static Extern

All of the external variables and functions in the proposed solution are of type extern and all except `expand_string( )` should be static. In C, any variable declared outside of any function is called an external variable. External variables are either of type extern or type static. Failure to explicitly declare the type results in an external variable of type extern.

The difference between static and extern external variables is that extern externals are available to functions compiled from other files, while static externals are available only in the file in which they are declared. Functions are also either extern or static with the same implications.

Failure to hide external variables and functions by declaring them to be static can lead to unanticipated side effects. All five of the external variables in this program and seven of the eight functions should be declared to be static. The `expand_string( )` function is the only function which should be of type extern.

Dynamic Memory Allocation

The statement of the problem requires that, "... an argument string can be of any length." Since the output from an argument string of constant tokens is the same length as the argument string, it follows that the output string must be permitted to be arbitrarily long. However, the proposed solution limits the output string to `EXPANSION_LENGTH_LIMIT` characters.

The C programming language is particularly well suited to dealing with arbitrarily long strings because of its library of dynamic allocation functions.

The program could use the `scan_TYPE_component( )` functions to determine the length of the output strings and then allocate the buffer space to hold the required length strings. Storage for the output strings could be allocated as it is needed in `recursively_expand( )`.

These program revisions can be made while retaining the original algorithm, and eliminating the constraints on the maximum string length and the maximum number of strings. These revisions can also reduce the number of variables and eliminate all of the external variables. The total number of lines of code does not change substantially.

Reentrant Design

The choice of external data in the proposed solution reduces the possibility of using `expand_string( )` from within a recursive algorithm. The program's author provides convincing arguments in favor of recursive algorithms, and identifies C's support of recursive functions as one of the primary reasons for selecting C as the most appropriate programming language. In addition, the author mentions that the resulting function will be used in an implementation language for describing circuits. And, language implementations frequently use recursive algorithms for parsing input. The inability to invoke `expand_string( )` from within a recursive algorithm can be regarded as a serious limitation in this context.

The restrictions on the use of `expand_string( )` from within a recursive algorithm can be removed by converting the external variables to automatic type, so that every invocation of `expand_string( )` creates its own copy of these variables.

The cost of making `expand_string( )` reentrant is one additional pointer in the parameter list for `recursively_expand( )`. This is a small price to pay for the reduction in "fuss."

Conclusions

Eric Hamilton has presented us with a well-crafted program that meets the algorithmic requirements of the problem but imposes serious constraints in application. The constraints include a limit on the number of strings that can be created, a limit on the length of the strings that can be created, and limitations on the use of the function from within a recursive parent function.

Writing code that can be plugged into someone else's program "without much fuss" means designing functions that impose the fewest restrictions on the input, and it means writing bulletproof code. Unanticipated side effects resulting from external variables unnecessarily declared as extern can create problems. Functions that are not needed outside of the file containing the primary function should be declared static to prevent other unanticipated side effects. Unnecessary and unchecked constraints on the size of objects can lead to unpredictable behavior.

The suggested improvements to this program emphasize the use of dynamic memory allocation and the replacement of external variables with automatic variables.

I coded these changes and found that they affected a relatively small percentage of the code and made no significant change to the total number of lines of code.

The analysis of the program and rewrite took about six hours, about as long as it took the author to write the code in first place! However, these changes eliminate the need to document the constraints and limitations in the original program.

Whenever I review someone else's program, I always look at the external data first, and, as is the case in this program, I usually find problems related to their use. External data reduces the independence of all of the functions that can access it, and external data frequently prevents the use as recursive program design. In addition, I have reviewed programs that use external data areas as data buffers by different functions at different times for different purposes. This design technique has frequently led to buggy code. Finally, using external data usually leads to programs with larger total memory requirement than would be the case if the data storage were dynamically allocated.

Don Colner
Polaris, Inc.
2202 Sherbrooke Way
Rockville, MD 20850

For Correspondence: Christopher J. Van Wyk, AT&T Bell Laboratories, Room 2C-457, 600 Mountain Avenue, Murray Hill, NJ 07974.