M0070

TO: Multics Repository

FROM: J. H. Saltzer

SUBJ: Character Handling and PL/I

DATE: June 30, 1966

In recent weeks much debate has occurred on the subject of character handling in the PL/I language, and on the relationship between PL/I, the ASCII character set, and the canonical form for ASCII character streams described in MSPM section BC.2.02. This memo is intended to summarize the basic issues involved and to offer a proposal for salvaging as much as practical from an awkward set of constraints. It is a result of conversations with many people, including M. D. McIlroy, R. Morris, V. Vyssotsky, F. J. Corbato, E. L. Glaser, R. M. Graham, and A. Evans.

The PL/I language includes a data type known as "character string" and a set of conventions for operators, literals, and built-in functions to manipulate this data type. Allocation of space for this data type, and certain of the built-in functions, are based upon simple character counts.

In the simplest possible interpretation of character strings, they are merely strings of 9-bit entities, with no knowledge of or interest in their contents. At present, this interpretation is used in the EPL compiler. This simple interpretation, while adequate in the absence of control characters, leaves some deficiencies when a character set with control characters is used. Two examples can serve to illustrate the types of problems encountered. First, consider the character string "A - dθ/dt" in which the "theta" is constructed of a capital "O" overstruck with a minus sign. Considering 9-bit characters as independent entities, the question "how many minus signs does this string contain" has the answer "two". The user who wishes to consider the second minus sign to be part of a theta" rather than a minus sign must explicitly program a check for a backspace character on each side of each potential minus sign. From this example, it is clear that there are users who need the ability to consider a "print position" (from BC.2.02) as an entity, rather than a 9-bit character.

A second example concerns input and output format statements. Suppose that the programmer is writing out tabular data consisting of character strings followed by integers, using a format such as "A10, X3, I5", expecting his character

strings to begin in column 1 of the page, and his integers to be in columns 14-18. If one of the character strings happens to contain an overstruck character, the integer on that line will be printed two columns to the left of where he expected. Even if the programmer realizes that some of his character strings might contain overstrikes, he is hard put to find a way to program past this problem.

A further complication introduced by the simple interpretation of character data by PL/I is that by use of the concatenation operator one can very readily construct strings which are not in canonical form.

Solutions

With this background, then, there have been proposed at least three different ways of handling canonical streams in PL/I programs.

1. Create a new data type, perhaps named "Print Position String", in which the object of manipulation is the print position.

2. Re-interpret the character data type so that wherever the term "character" appears the term "print position" is understood.

3. Using the simple interpretation of character strings, add a list of subroutines to the library to replace the PL/I built-in functions when working with canonical streams which may contain control characters. This proposal has roughly the effect of implementing the first proposal "outside of the language."

The first two proposals, while attractive, cost an obvious price in lost compatibility with other PL/I compilers. They also have a more fundamental shortcoming, namely that the operations which are appropriate for a print position data type are far from obvious; one can only say with some certainty that the PL/I character manipulation operations are not adequate for dealing with print positions. For example, the built-in function "length" is not immediately useful in counting print positions if a character string consists of more than one line.

It is therefore suggested here that the third alternative be followed, and that a minor change to the canonical form be made so that a minimum of special concern is required of the programmer not using control characters. The remainder of this memo describes 1) changes to canonical form, and

2) a suggested list of library subroutines needed to handle canonical strings as a sequence of print positions rather than a sequence of characters.

Relative Tabulate Characters

MSPM Section BC.2.01 originally proposed that horizontal blank space be represented in the canonical form exclusively by a Relative Horizontal Tabulate charater, consisting of a special ASCII control character, followed by a space count. The reasons for this proposal were two-fold:

1. To permit trivial comparison of two strings which differ only in the amount of space appearing between two graphics. (And thus to encourage the notion that typed-in horizontal space is simply horizontal space, no matter how much of it a sloppy typist uses.)

2. To provide a measure of code compression by replacing repeated blanks with a two-character entity.

While these objectives are still valid, a number of problems arise if ordinary PL/I character-string operators are used on strings containing blank space represented by relative horizontal tabs:

1. The single space is represented by a "space" character rather than RHT 1. A user dealing with the raw characters will observe this singularity.

2. In order to analyze a string of characters with the PL/I string operators, it is necessary to do some extra work. Two approaches are either to "decanonicalize" the string by having a subroutine replace RHT's with blanks, or use "UNSPEC" or mis$-matched declarations across calls to analyze the relative tabs.

The decanonicalizer would probably result in files being written into the file system containing blanks as well as RHT's. Analysis of RHT's by subroutine requires quite a bit of extra effort by the routine programmer. Without going to a third alternative of completely abandoning PL/I built-in character operations as "implementation dependent", the simplest approach is to instead abandon the relative horizontal tab. For similar reasons, the relative vertical tab is also abandoned.

Horizontal space is then represented in the canonical form by the appropriate number of "space" characters, vertical space by the appropriate number of "New Line" characters.

Canonical String Manipulation Subroutines

The following subroutines are suggested as a starting point for a complete library of operators for canonical strings. The fundamental unit of manipulation is the print position, as defined in BC.2.02. Except for the first subroutine, it is assumed that all argument strings are canonical.

1. Canonicalizer

```
y = canonicalize (a)
```

The value of canonicalize is a varying string equal to the result of converting the string a to canonical form. (Note that no "decanonicalizer" is necessary, since a canonical string is a perfectly ordinary character string.)

2. Print Position Extraction

```
y = pp_substr (a,n,m)
```

The value of pp_substr is a varying string consisting of the nth through the (n+m-1)th print positions of the canonical string a. Note that the result of pp_substr is a canonical string. Consider, for example, the canonical string x = "a⟨HLR⟩256⟨HLF⟩" (representing a^{256}) the result of pp_substr(x,3,1) would be "⟨HLR⟩5⟨HLF⟩".

3. Print Position Inserter

```
call pp_subst (y,a,n,m)
```

The string y is substituted into the string a in the nth through (n+m-1)th print positions of a.

4. Print position counter

```
k = pp_length (a,j)
```

The value of pp_length is the number of print positions in the jth line of the string a.

5. Line counter

```
k = line_count(a)
```

The value of line_count is the number of complete lines in the string a. The string "abc" is zero

lines; the string "abc⟨NL⟩" is one line. The string "abc⟨NL⟩def" is still one line, but "abc⟨NL⟩def⟨NL⟩ is two lines. In general, the line count equals the number of new line characters in the string.

6. Check for line fragment

```
b = line_fragment(a)
```

The value of line_fragment is a 1-bit string with value '0' b if the string a ends with a New Line character. Otherwise its value is '1' b.

7. Line Extractor

```
y = line_substr(a,n,m)
```

The value of line_subst is a varying string containing the nth through the (n+m-1)th lines of the string a.