Draft

CGTM 106
David Gries
March 1968

Compiler Implementation System

Strings and input-output

This note describes the uses of strings in compiler-compiler language. Strings are introduced only to make input-output easier (error messages, etc.). This note supercedes any mention of strings in earlier notes.

String manipulation

1. Introduction

Strings are intended to be used only for input-output functions — changing the input source program, inserting or deleting characters from it, printing the source program and error messages, etc. Internally, source program characters and atoms themselves are not used since they are represented by 16-bit (BYTE2) entities; thus string manipulation should not be necessary.

2. String variables

The lengths of string variables are fixed at meta-compile time and may not exceed 255.

A simple string may be defined by

<declaration> ::= STRING [(<unsigned integer>) <identifier> | <identifier> .= <string constant>]

With the first alternate, the <unsigned integer> specifies the length. In the second case, the <identifier> is initialized to <string constant> (see 3a); the length is the number of characters in the <string constant>.

Tables (arrays) of strings are STATIC only, and are declared by

<declaration> ::= STRING (<unsigned integer>$_1$) TABLE (<unsigned integer>$_2$)

<identifier> .= [<string constant> [,]* |]

::= STRING TABLE <identifier> .= <string constant> [,]*

In the first case, <unsigned integer>$_1$ specifies the length of each string in the static table, while <unsigned integer>$_2$ specifies the number of strings. If n <string constant>s appear, the first n elements are initialized to these <string constant>s.

In the second case, the number of strings in the table is the number of string constants; the length of each element is the length of the corresponding <string constant> to which it is initialized.

3. String assignments

A string assignment has the form

<assignment> ::= <string variable> .= <string>

<assignment> ::= SUBSTR (<string variable>,<integer expression>$_1$ [,<integer expr.>$_2$]) .= <string>

In the second case, we must have length (<string variable>) $\geq$ <integer expr>$_1$ + <integer expr>$_2$.

In this case, <integer expression>$_2$ characters of <string variable>, beginning with the <integer expression>$_1^{th}$, are replaced by the first <integer expression>$_2$ characters of <string>. If <string> has too few characters, blanks are added on the right; if too many, only the leftmost are used.

The default option for <integer expression>$_2$ is (length (string variable) - <integer expression>$_1$ + 1).

The first form is equivalent to

SUBSTR (<string variable>, 1) .= <string>

4. <string>s

The following are <string>s.

a) A <string constant> — of the form 'sequence of EBCDIC characters'

In the sequence the character ' must be representedby two apostrophes. Thus"B''C'' "is the string " B'C' ".

b) A string constant of the form X' (pairs of hexcharacters 0-9, A-F) '. The value is the characters described by each pair of hex characters.

c) TEXT (<expression>)

The <expression> must be the internal representation (BYTE2) of some atom or a pointer to some element of a DICT or the internal dictionary. The value of this <string> is the string of characters forming the atom on which the element is chained.

d) A <syn identifier> – synonym for some source language atom.

e) A variable declared as STRING.

f) SUBSTR (<string>, <integer expression>$_1$, <integer expression>$_2$)

The value is the <integer expression>$_2$ characters of <string>, starting with the <integer expression>$_1{}^{th}$ character.

g) BITTEXT (<expression>)

The value is the binary value of the expression as it appears in memory. (Thus BITTEXT (B'11010') is equal to '11010'.

h) OCTTEXT (<expression>)

The value is the octal value of the expression: OCTTEXT (B'11010') is '32'.

i) HEXTEXT (<expression>)

The value is the hex value of the expression: HEXTEXT (B'11010') is'1A'.

j) INTTEXT (<expression>)

The expression is converted to a decimal integer and changed to character form.

k) FLTEXT (<expression>)

The value is the <expression> in some fixed, decimal floating point representation.

ℓ) <string>$_1$ CAT <string>$_2$

The result is the string <string>$_1$ concatenated with the string <string>$_2$.

5. System names

&INPUT is a string variable which always contains the complete current source program line, as read in. It may not be assigned another value.

&SCANNER is a string variable containing the current source program line being used by the scanner (lexical analyzer). As characters are "used up" to form atoms they are deleted from &SCANNER.

&POSIT, type HWI, always contains the current position of the scanner in &SCANNER from the beginning of the original line. Thus if three characters on the current line have been "used up" to form atoms, &POSIT has the value 4.

&LINE, type HWI, contains the current output line number (the first card or line is number 1).

Strings can be added to the current beginning of the line in &SCANNER by the statement

&INSSCAN (<string>)

&POSIT is counted back the appropriate number of characters. This may be useful in error recovery procedures.

6. Output

One adds a string of characters to the end of the current line of

output by the statement

$\mathcal{E}$OUTPUT (<string>)

The statement

$\mathcal{E}$OUTPUT (<string>$_1$,<string>$_2$, ... ,<string>$_n$)

is equivalent to

$\mathcal{E}$OUTPUT (<string>$_1$); ... ; $\mathcal{E}$OUTPUT (<string>$_n$)

When the current output line is filled up, it is written out and a new one started. Alternatively, one can cause the current line to be written out by executing

$\mathcal{E}$OUTPUT .

Execution of the statement

$\mathcal{E}$OUTDESCR(D)

where D is a DESCRIPTOR (see CGTM #104) or a pointer to a DESCRIPTOR cuases the current line to be written out and the DESCRIPTOR D to be written out in some as yet undetermined format.