String Processing Languages

Ralph E. Griswold
Department of Computer Science, The University of Arizona, Tucson, AZ 85721

and

David R. Hanson
Department of Computer Science, Princeton University, Princeton, NJ 08544

Research Report CS-TR-306-91

February 1991

Abstract

In most traditional programming languages, character strings are arrays of characters, and string processing is programmed with low-level array operations, such as indexing. In high-level string-processing languages, strings are treated at a higher conceptual level. Strings are first-class values in these languages, and most offer a rich set of operations that manipulate strings as atomic values. Examples include concatenation, substring identification, transformation, and pattern matching. This report surveys some of the past and present string-processing languages, emphasizing their approaches to string processing and their facilities that deal with strings. Included are brief descriptions of Comit, the SNOBOL languages, and Icon. This report will appear in the third edition of the *Encyclopedia of Computer Science and Engineering*, to be published by Van Nostrand Reinhold.

Principles

In programming contexts, the term *string* usually refers to a sequence of characters. For example, ABC is a string of three characters. Strings are more prevalent in computing than is generally realized. In most cases, computer input is in the form of strings, e.g., commands entered at a terminal. Similarly, computer output is in the form of strings; printed lines are simply strings of characters. In spite of this fact, string processing has received comparatively little attention.

Strings and String Processing

The facilities of the most widely used and well-known programming languages are concentrated on numerical and business data processing. However, a substantial amount of string processing is performed. For example, compilers accept strings as input, analyze them, and usually produce other strings as output. Command interpreters analyze command strings and perform appropriate actions. These kinds of programs are used heavily, so they must be extremely efficient. For this reason, they are usually written in systems programming languages, e.g., C, rather than in higher-level string-processing languages.

Nevertheless, higher-level string-processing languages offer many advantages for solving complex problems. Examples of such problems are language translation, computational linguistics, symbolic mathematics, text editing, and document formatting.

Developers of string-processing languages are in a less well-defined position than developers of numerical processing languages. While mathematical notation for numerical computation has developed over centuries, and its current form is widely known and fairly well standardized, string processing is a new area. There is no general agreement on what operations should be performed in string processing, nor is there a standard notation. The developers of string-processing languages started largely without conventions. As a result, notation, program structure, and approach to problem formulation are often radically different from those of more conventional programming languages.

Operations on Strings

While there were no generally agreed-upon string operations when string-processing languages were first developed, four operations have achieved general acceptance: concatenation, identification of substrings, pattern matching, and transformation of strings to replace identified substrings by other strings.

Concatenation (sometimes called "catenation") is the process of appending one string to another to produce a longer string. Thus, the result of concatenating the strings `AB` and `CDE` is the string `ABCDE`. This operation is a natural extension of the concept of a string as a sequence of characters. A *substring* is a string within another string. For example, `BC` is a substring of `ABCDE`.

The most important string operation is pattern matching. Stated in general terms, *pattern matching* is the process of examining a string to locate substrings or to determine if a string has certain properties. Examples are the presence of a specific substring, substrings in certain positions, substrings in a specified relationship to each other, and so on. *Transformation* of strings is typically accomplished in conjunction with pattern matching, using the results of pattern matching to effect a replacement of substrings.

The language descriptions below emphasize approaches to string processing and the major facilities that deal with strings. No attempt has been made to describe these languages completely; details can be found in the references.

Languages

Comit

Comit [11], designed in 1957–58, was the first string-processing language. It was motivated by the need for a tool for mechanical language translation. Comit strongly reflects these origins and is oriented toward the representation of natural languages.

Basic Concepts

In Comit, unlike most other string-processing languages, a string is composed of *constituents*, which may consist of more than one character. Thus, a word composed of many characters may be a single constituent in a string. A string is written as a series of constituents separated by + signs, e.g.,

```
FOURSCORE + AND + SEVEN + YEARS + AGO
```

The character - represents a space (blank). Thus, to include spaces between words, the string above becomes

```
FOURSCORE + - + AND - + SEVEN + - YEARS + - AGO
```

All characters other than letters have syntactic meaning. A star (asterisk) in front of a character other than a letter indicates that the character is to be taken literally rather than for its syntactic meaning. For example,

```
33 ARE IN THE TOP 1/2.
```

is written

```
*3*3 + - + ARE + - + IN + - + THE + - + TOP + - + *1*/*2*.
```

Attention focuses on a *workspace*, which contains the string currently being processed. There are 128 *shelves*, any of which may be exchanged with the workspace to change the focus of attention. Thus, there may be at most 129 distinct strings in a program at any one time.

Comit programs are a sequence of rules, each of which has five parts:

name left-half = *right-half* // *routing goto*

The *name* identifies the rule. The *left-half* is a pattern applied to the workspace, and the *right-half* specifies processing to be performed on the portion of the work space matched by the *left-half.* The *routing* performs operations other than pattern matching. If a rule has no routing field, the slashes are not required. The *goto* controls program flow.

Pattern Matching

The left-half may specify full constituents as written in a string, a specific number of constituents of unspecified value, an indefinite number of constituents, an earlier constituent referenced by its position in the left-half, and so on. A full constituent is written as it is in a string. Other left-half constituents are represented by special notations. For example: $\$n$ matches n consecutive constituents, regardless of their value; \$ matches any number of constituents. The integer n matches the same string that the nth constituent of the left-half matched. For example, the left-half

```
THE + $1 + $ + 2
```

has four constituents: the characters THE, followed by any single constituent, followed by any number of constituents until one is encountered that is the same as the one matched by the second constituent, namely, $1. Pattern matching is left to right. Left-half constituents must match consecutive constituents in the workspace.

If the workspace contains

$$\underbrace{\texttt{THE}}_{1} + \underbrace{\texttt{FIRST}}_{2}$$

$$\underbrace{\texttt{+ MAN + IN + LINE + IS + SERVED}}_{3} + \underbrace{\texttt{FIRST}}_{4}$$

the match for each of the constituents is as shown. Note that the fourth constituent of the left-half matches the same constituents as the second constituent of the left-half. The third constituent of the left-half consequently matches the intervening five constituents. When a match occurs, workspace constituents are associated with the left-half constituents they matched and are subsequently referenced by the number of the corresponding left-half constituent.

The right-half may contain full constituents and integers that correspond to the constituents of the left-half. The matched portion of the workspace is replaced by constituents specified in the right-half. Continuing the example above, the rule

```
THE + $1 + $ + 2 = 1 + SECOND + 3 + 4
```

transforms the workspace into

```
THE + SECOND + MAN + IN + LINE + IS + SERVED + FIRST
```

Other Facilities

The routing part of a rule permits operations that cannot be performed in the right-half. Examples are exchange of the workspace with a shelf, movement of constituents between the workspace and shelves, printing the workspace, reading data into the workspace, and so on.

The goto part of a rule controls program flow. Control may be transferred to a named rule, back to the same statement for execution again, to the next statement, and so on.

Loops may be programmed in a number of ways. One conditional operation is left-half matching, which may fail. For example, the left-half $10 would fail to match the workspace given above because the workspace does not contain ten constituents. When a left-half fails to match, the remainder of the rule is not performed and control passes to the next rule in line. Special notations are used for names and gotos in writing loops. A * may be used for the name of a rule that needs no other specific identification. A * in the goto indicates that control is to be transferred to the next rule in line. A / in the goto indicates that control is to be returned to the present rule if it is executed successfully. Thus,

```
    * THE = /
```

which has a blank right-half and a / in the goto, removes all occurrences of THE from the workspace. When the left-half finally fails to match, execution continues with the next rule in line.

Status

The current version of Comit is Comit II, which has been implemented on the IBM 7000 series and the IBM 360/370. Because of its early origin, Comit lacks a number of features that are available in more recently developed languages.

SNOBOL and SNOBOL3

The first SNOBOL (string-oriented symbolic language) language was designed and implemented in 1962–63. The major motivation was the need for a general-purpose language for string processing. Manipulation of symbolic mathematical expressions was also an important consideration.

Basic Concepts

In SNOBOL, unlike Comit, a string is simply a sequence of characters. Enclosing quotation marks delimit the string, but are not part of the string. An example is

```
    'FOURSCORE AND SEVEN YEARS AGO'
```

Such a string is said to be specified *literally*. Strings may be assigned to names for subsequent reference, e.g.,

```
FIRST = 'MORGAN'
```

assigns the string MORGAN to the name FIRST. There is no limit to the number of distinct strings. Storage management is automatic; there are no declarations. Concatenation is denoted by the juxtaposition of strings. Such strings can be given literally or as the value of names, e.g.,

```
FULLNAME = FIRST '␣SMITH'
```

assigns the string MORGAN␣SMITH to the name FULLNAME. The blank, shown here as ␣ for clarity, is simply a character like any other.

A SNOBOL program consists of a sequence of statements. There are three kinds of statements: assignment, pattern matching, and replacement. The respective forms are

label	*subject* = *object*	*goto*
label	*subject pattern*	*goto*
label	*subject pattern* = *object*	*goto*

An optional *label* identifies the statement. The *subject* provides the focus for the statement and is the name on which operations are performed. The *goto* controls program flow and is optional. An assignment statement assigns a value to a name. A pattern-matching statement examines the value of a name for a *pattern*, and a replacement statement modifies that part of the subject matched by the pattern.

Pattern Matching

Patterns in SNOBOL consist of a sequence of components. There are two types of components: specific strings and *string variables.* A specific string may be given literally or referred to by name. A string variable is indicated by delimiting asterisks, which bracket a name. There are several types of string variables. An *arbitrary string variable* can match any string. It is similar to the Comit $ notation, except that whatever the string variable matches is assigned to the name between the asterisks. Pattern matching is left to right, and components of the pattern must match consecutive substrings of the subject. For example, in

```
Z 'T' *FILL* 'N'
```

the value of Z is matched for any string that begins with a T and ends with an N. The substring between the T and N is assigned to the name FILL. If the value of Z is TEEN, the value assigned to FILL is EE.

A *balanced string variable* matches a string that is properly balanced with respect to parentheses like an ordinary mathematical expression. A *fixed-length string variable* matches any string of a specific length and are indicated by a / and a quoted number following the name. For example,

```
TEXT ',' *C/"1"*
```

examines the value of TEXT for a comma and assigns the character following the comma to C.

Replacement is a combination of pattern matching and assignment in which the matched substring is replaced by the object. The statement

```
FULLNAME 'SMITH' = 'JONES'
```

replaces the substring SMITH by JONES and consequently changes the value of FULLNAME to MORGAN␣JONES.

Indirect Referencing

A string may be computed and then used as a name. A $ placed in front of a string uses the value of that string as a name. For example, the statements

```
X = 'NUM'
N = '3'
HOLIDAY = XN
$HOLIDAY = 'EASTER'
```

first assign the value NUM3 to HOLIDAY and then assign the value EASTER to NUM3. The indirect referencing operator, similar in concept to indirect addressing in assembly language, provides a way of constructing names of data during execution.

Other Facilities

Input and output take place using specially designated names as subjects. Arithmetic facilities are rudimentary, e.g., integer arithmetic on strings of digits.

The goto part of a statement controls program flow. Gotos can be unconditional to a labeled statement, or conditional on the success or failure of pattern matching. Loops are programmed using the conditional nature of pattern matching.

Status

SNOBOL was superseded by SNOBOL3 in 1965. SNOBOL3 is similar to SNOBOL, but has several additional features, including a number of built-in functions and a facility for programmer-defined, recursive functions. SNOBOL3 was in turn superseded by SNOBOL4 in 1967.

SNOBOL4

SNOBOL4 [8] is a natural descendant of earlier SNOBOL languages and is based on many of the same ideas and approaches to string processing. SNOBOL4, however, introduced a number of new concepts. The most important, from a string-processing poing of view, are those dealing with pattern matching.

Patterns

In Comit and the earlier SNOBOL languages, different types of patterns are indicated by specific notations. In SNOBOL4, patterns are data objects that are constructed by functions and operations. Consequently, quite complicated patterns can be built piecemeal.

There are two basic pattern-construction operations: alternation and concatenation. The alternation of two patterns is a pattern that will match anything either of its two components will match. The concatenation of two patterns is a pattern that will match anything its two components will match consecutively. Alternation is represented by a vertical bar and concatenation by a blank, e.g.,

```
PET = 'CAT' | 'DOG'
PETKIND = PET '-LIKE'
```

The pattern `PET` matches either of the strings `CAT` or `DOG`, and `PETKIND` matches anything `PET` matches followed by the string `-LIKE` (i.e., `CAT-LIKE` or `DOG-LIKE`).

Pattern-valued functions generalize the concept of patterns and avoid special notations for each type. For example, the value returned by `LEN(`n`)` is a pattern that matches n characters, and the pattern returned by `TAB(`n`)` matches a substring through the nth character of the subject string. For example,

```
OPER = TAB(6) 'X'
```

creates a pattern that will match any string containing an X as its seventh character. Other pattern-valued functions create patterns that match any one of a number of specific characters, search for specific characters, and so on. Examples are SPAN('0123456789'), which matches a substring consisting only of digits, and BREAK(';,'), which matches the substring beginning at the current position up to the next comma or semicolon.

As in SNOBOL, pattern matching is left to right, and components must match consecutive substrings of the subject string. When a component fails to match, alternative matches are attempted. If no alternative is specified, the pattern-matching process backs up to earlier, successfully matched components, seeking other ways in which the entire pattern match can succeed. Conceptually, the pattern-matching process manipulates a cursor, which is an imaginary marker in the subject string indicating the current position of the match. Movement of the cursor is implicit, not under direct control of the programmer, although in some patterns there is a direct correlation. Thus LEN(3) moves the cursor to the right three characters. The cursor cannot be moved to the left by a successful match.

Names may be attached to components of patterns so that when the component matches a substring, that substring is assigned to the name. Attachment is indicated by the binary $ operator, e.g.,

```
    HEAD = LEN(7) $ LABEL
```

constructs a pattern that matches seven characters. The seven characters, when matched, are assigned to LABEL, so

```
    CARD HEAD
```

assigns the first seven characters of the value of CARD to LABEL. If the match fails (because CARD is less than seven characters long), no assignment is made to LABEL.

Another aspect of pattern matching is the ability to modify the pattern during matching depending on substrings matched by earlier components. Evaluation of an expression in a pattern may be deferred by prefacing the expression by *. The expression is then left unevaluated until it is encountered in pattern matching. An example of the power of this facility is given by

```
    LIT = ('"' | "'") $ C BREAK(*C) . STRING LEN(1)
```

When LIT is used in pattern matching, the argument of BREAK is not evaluated until after the first part of the pattern has matched. The pattern matches a single or double quote and assigns it to `C`. The remainder of the pattern matches everything up to the next occurrence of character just assigned to `C`, assigns that substring to `STRING`, and then `LEN(1)` matches the second quote. Thus, `LIT` matches literal string constants as used in many programming languages.

Other Facilities

Other string processing facilities include alphabetical comparison of strings, mappings from one set of characters to another, and deletion of trailing blanks. Earlier SNOBOL languages were purely string processing languages; SNOBOL4 includes many types of data. In addition to types such as integer and real, SNOBOL4 includes arrays as data objects, tables that provide associative look-up features, and a facility for defining record types during execution. In many cases, it is possible to perform data-type conversions between various types of data. It is possible to convert a string into program statements during program execution, and hence to modify or extend the program while it is running. SNOBOL4 is actually a general-purpose language that strongly emphasizes string processing and contains a number of exotic features.

Status

SNOBOL4 is a widely used and generally available string-processing language. It and a number of dialects have been implemented on a wide range of machines from personal computers to mainframes.

Icon

The major emphasis in pattern matching in the SNOBOL languages, as in other string-processing languages, is on the *specification* of patterns that analyze strings. There is little facility for indicating *how* the matching is accomplished or for describing the synthesis of new strings from the results of pattern matching.

In many cases, this bias toward pattern specification is useful; it frees the programmer from the necessity of spelling out too much detail concerning the actual matching. This is especially the case in SNOBOL4, in which the process of matching embodies a powerful search and backtrack algorithm that is particularly complex and obscure.

In other cases, however, programming tasks may fall outside the capabilities of the pattern matching facility. Faced with this dilemma, programmers resort to inefficient or obscure techniques that are typically unrepresentative of the capabilities of the language as a whole. This situation is due largely to the inextensibility of the pattern matching facility. In SNOBOL4, for example, the pattern matching facility is not as extensible as is the rest of the language. While there is a facility for programmer-defined functions and datatypes, there is no facility for programmer-defined *matching* procedures, i.e., procedures, which are invoked during matching, that describe how a particular pattern is to be matched. This deficiency can be better understood by considering the pattern assigned to `HEAD` above:

```
    HEAD = LEN(7) $ LABEL
```

`LEN(7)` constructs and returns a pattern that, when applied, attempts to advance the cursor by 7 characters. `LEN` itself plays no role in the matching — it merely constructs a data object that contains an indication of the action to be taken during pattern matching. It is this latter component of the pattern that corresponds to the matching procedure and that cannot be defined by the programmer.

SNOBOL4 and its variants suffer a common problem: They are each, in reality, composed of two languages — a basic language and a pattern matching language [6]. In each language, the programmer is burdened with the construction of pattern matching "programs." This corresponds to construction of a pattern, which is subsequently applied, or to the construction of the set of procedures, which eventually cooperate during pattern matching. This two-step process — pattern construction and pattern application — is due largely to the central role of patterns as distinguished objects in string processing languages. It is the elimination of patterns, but not of pattern matching, that differentiates the newest string processing language, Icon, from its predecessors.

Icon [5], developed in late 1970s, has a number of relatively low-level lexical primitives, some of which are related to patterns in SNOBOL4. Icon also has control structures and a goal-directed evaluation mechanism that make pattern matching — called string scanning in Icon — an integral part of the language. The central feature of Icon is this evaluation mechanism, which embodies a search and backtrack algorithm similar, but simpler, than that used in SNOBOL4 pattern matching. An important aspect of this mechanism is that it pervades the entire language, instead of being restricted to a component of the language. The combination of the lexical primitives and the evaluation mechanism yields string scanning capabilities comparable to those of SNOBOL4.

String scanning in Icon is accomplished in a manner that appears similar to SNOBOL4 but does not involve anything like pattern construction. The expression s `?` e establishes s as the subject to which string processing operations in e apply. The expression e typically includes string analysis operations, but may include *any* Icon operation. A *scanning environment* is characterized by a pair of implicit variables {`subject`, `pos`}; `subject` is the string to which scanning operations apply and `pos` is a location with the subject and usually changes as the subject is analyzed. The expression s `?` e establishes a new scanning environment $\{s, 1\}$, and then evaluates e. After evaluating e, the previous scanning environment is restored.

Some of the scanning operations in Icon operate on the position in the absence of other specifications. An example is `move(`n`)`, which attempts to advance the position by n characters. If the advancement is successful, `move` returns the n-character substring between the initial and final positions. For example,

```
line ? write("[", move(7), "]")
```

writes the first 7 characters of `line` enclosed in brackets to the output. The equivalent SNOBOL4 program

```
HEAD = LEN(7) $ LABEL
LINE  HEAD
OUTPUT = "[" LABEL "]"
```

first constructs the necessary pattern and assigns it to `HEAD`, then applies this pattern to `LINE`, which causes the first 7 characters to be assigned to `LABEL`, and finally writes the desired result.

This simple example illustrates an important aspect of string scanning in Icon: `move` does not construct a pattern, but simply carries out the analysis in the current scanning environment. The SNOBOL4 equivalent involves construction of a pattern, followed by its application, and finally the output of the desired result.

Another important advantage resulting from the integration of string processing with the rest of Icon is that any language operation can be performed during string scanning. An example is

```
line ? while t := t || move(1) || "."
```

which produces a string `t` containing the characters of `line` separated by periods. The `||` operator denotes string concatenation, and `while` repeatedly evaluates

```
t := t || move(1) || "."
```

until it fails, which occurs when `move(1)` is invoked at the end of the subject string. Note the use of a standard control structure, `while`, within the `?` expression. To accomplish the same thing in SNOBOL4 requires the separation of the analysis of the subject and the synthesis of the result since there is no provision for using arbitrary constructs within a pattern. Thus, the SNOBOL4 equivalent requires two statements:

```
LOOP TEXT LEN(1) $ C =     :F(DONE)
     T = T C "."           :(LOOP)
DONE
```

If the pattern in the first line fails to match, control is transferred to the line labeled `DONE`; otherwise the matched character and a period are appended to `T`, and control is returned to the line labeled `LOOP`.

String *synthesis* often accompanies string scanning. In the example above, `t` is synthesized during scanning, and it is `t` that is the result of interest. In some cases, the result of interest can be returned as the value of the scanning expression. The result of *s* `?` *e* is the result of *e*, so both of the expressions

```
line ? write("[", move(7), "]")
write("[", line ? move(7), "]")
```

the same output.

The function `move(`*n*`)` is called a *matching function* because it returns the substring of the subject that is "matched" as a result of changing the position. Another matching function is `tab(`*i*`)`, which moves to position *i* in the subject and returns the substring between the old and new positions. For both `move` and `tab`, the new position can be to the left of the old position.

Lexical functions return positions in the subject instead of substrings in the subject. For example, `find(`*s*`)` returns the position of the string *s* in the subject following the current position, so the output of

```
"Icon is a programming language" ? write(find("program"))
```

is 11. Likewise, `upto(s)` returns the position of any of the characters in string s, and `many(s)` returns the position following the longest possible substring containing only characters in s starting at the current position.

It is important to note that functions like `many` return positions, but the specific values of those positions are rarely important. Positions are used most often as arguments to matching functions like `tab`. For example,

```
line ? while tab(upto(&letters)) do
   write(tab(many(&letters)))
```

writes the "words" in `line`. The value of the keyword `&letters` is a string containing all of the upper- and lower-case letters. The expression `tab(upto(&letters))` advances the position up to the next letter, and `tab(many(&letters))` matches and returns the word, which is passed to `write`. The `while` loop terminates when `tab(upto(&letters))` fails because there are no more words in `line`.

Most changes to the scanning environment, e.g., changing the position, are made implicitly by matching functions. Explicit reference to the scanning environment can be made through the keywords `&subject` and `&pos`, e.g.,

```
&pos := 1
```

sets the scanning position to `1`. This assignment is equivalent to `tab(1)`. Likewise,

```
&subject := read()
```

changes the subject to the next line of input. Assignments to `&subject` cause `&pos` to be set to `1`. It is usually undesirable to access or change the subject and position explicitly. However, doing so is necessary when writing matching procedures to augment the built-in repertoire of matching functions. See Reference [3] for details.

Icon's alternation expression resembles alternation in SNOBOL4: e_1 `|` e_2. The important difference is that while the SNOBOL4 alternation operator constructs a pattern, alternation in Icon simply carries out the operation directly. The operation is similar to that performed during pattern matching in SNOBOL4 when the pattern constructed by `P1 | P2` is applied.

In the Icon expression e_1 `|` e_2, e_1 is evaluated first and if that evaluation succeeds, the value if e_1 is the result of the entire expression. If, however, evaluation of e_1 fails, the result is the result of evaluating e_2. Another way in which e_2 can be evaluated is if the entire expression is used in a context where the value of e_1 is unacceptable. An example is

```
move(10 | 5)
```

The expression `10 | 5` has two literal subexpressions and the first `10`, succeeds. Suppose, however, that the subject is only six characters long. In this case, `move(10)` fails. This causes the re-evaluation of `10 | 5`, which yields the value `5`. This time, `move(5)` succeeds. Note that

```
move(10 | 5)
```

is equivalent to

```
move(10) | move(5)
```

which corresponds to the SNOBOL4 pattern

```
LEN(10) | LEN(5)
```

Note, however, that SNOBOL4 has no direct counterpart to `move(10 | 5)`. Alternation in SNOBOL4 is restricted to specific contexts; alternation in Icon may be used anywhere an expression may be used.

In Icon, operations that have the capacity for producing alternative values are required by the context in which they appear are called *generators*. In addition to alternation, many of the low-level lexical primitives are generators whose behavior when used in string scanning is designed to facilitate string processing. For example, `find(`*s*`)` is capable of generating all of the positions at which *s* appears in the subject. If only one value is needed, only one is generated, so the output of

```
"a fish is a fish is a fish" ? write(find("fish"))
```

is `3`. Additional values are generated as demanded by the context in which `find` is used; for example, in

```
"a fish is a fish is a fish" ? write(find("fish") > 20)
```

the first value produced by `find` is `3`, which is less than 20. The comparison (`>`) fails, which causes `find` to be resumed. It produces `13` and the comparison again fails. Finally, `find` produces `23` and the comparison succeeds. A successful comparison returns its right operand, so the output is `20`.

Icon's procedure mechanism of allows the construction of programmer-defined generators. This capability corresponds to defining programmer-defined matching procedures in SNOBOL4. Generators are not limited to the string processing aspects of Icon, but is meaningful for many operations. Generators allow a more natural expression of some constructions than is possible in most other programming languages. It is often possible to express constructions more concisely and closer to the way programmers think in mathematical and natural languages. For further information about this aspect of Icon, see Reference [7]. Reference [4] describes an implementation of Icon in detail.

Status

Icon has been implemented on a wide range of computers, from personal computers to large-scale mainframes. It is the most widely used and generally available high-level string processing language.

Other String-Processing Languages

Ambit [10], developed in 1964, is a string-processing language oriented toward algebraic manipulation. Ambit is similar in many respects to Comit and the SNOBOL languages. However, its strings are parenthesized expressions that correspond to tree structures, and they are implemented as fully linked trees. In Ambit, unlike most other string-processing languages, two strings are considered equivalent even if they differ in the position and number of blanks they contain. A *basic replacement* rule consists of a *citation*, specifying a pattern, and a *replacement*, which effects a transformation on the string under consideration. The citation may match only one way; the replacement rule must be unambiguous. An important aspect of Ambit pattern matching is the explicit reference to pointers, which identify specific positions in strings. There are three variants of Ambit: Ambit/S for manipulating trings, Ambit/G for manipulating general data structures, and Ambit/L for list processing.

Convert [9] is an extension of Lisp, incorporating pattern-matching and transformation operations. There are a number of fundamental patterns and facilities for constructing most complicated ones. The function `RESEMBLE` applies patterns to strings, and `REPLACE` performs transformations using skeletons that specify the structure of the replacement. A rule consists of a pattern and a skeleton. Convert applies the pattern to a string. If a "resemblance" is found, values of relevant parts are identified and substituted into the skeleton to effect the conversion.

Axle [1], like Comit, has a workspace that is the focus of attention for pattern matching and replacement. Axle has *assertion tables*, which specify patterns. These specifications may be recursive. *Imperative tables* specify patterns to be matched and corresponding replacements. A pattern-matching procedure determines which imperative is applicable. Axle has *markers*, which may be positioned in the work space. These markers may be used to avoid reprocessing previously transformed parts of the workspace.

Panon [2] is based on generalized Markov algorithms and includes a number of pattern-matching facilities and rules for transforming strings. A Panon program is itself a string, and hence susceptible to self-modification.

References

[1] Kenneth Cohen and J. H. Wegstein. AXLE: An axiomatic language for string transformations. *Communications of the ACM*, 8(11):657–661, November 1965.

[2] A. Caracciolo Forino. String processing languages and generalized Markov algorithms. In *Proceedings of the IFIP Working Conference on Symbol Manipulation Languages*, pages 141–206, Amsterdam, 1968. North Holland.

[3] Ralph E. Griswold. String scanning in the Icon programming language. *The Computer Journal*, 33(2):98–107, April 1990.

[4] Ralph E. Griswold and Madge T. Griswold. *The Implementation of the Icon Programming Language.* Princeton University Press, Princeton, NJ, 1986.

[5] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language.* Prentice Hall, Englewood Cliffs, NJ, second edition, 1990.

[6] Ralph E. Griswold and David R. Hanson. An alternative to the use of patterns in string processing. *ACM Transactions on Programming Languages and Systems*, 2(2):153–172, April 1980.

[7] Ralph E. Griswold, David R. Hanson, and John T. Korb. Generators in Icon. *ACM Transactions on Programming Languages and Systems*, 3(2):144–161, April 1981.

[8] Ralph E. Griswold, James F. Poage, and Ivan P. Polonsky. *The SNOBOL4 Programming Language.* Prentice Hall, Englewood Cliffs, NJ, second edition, 1971.

[9] Adolfo Guzman and Harold V. McIntosh. Convert. *Communications of the ACM*, 9(8):604–615, August 1966.

[10] Michael S. Wolfberg. Fundamentals of the Ambit/L list-processing language. *SIGPLAN Notices*, 7(10):66–75, October 1972.

[11] Victor H. Yngve. *Computer Programming with COMIT II.* MIT Press, Cambridge, MA, 1963.