Chapter 9

Aggregates

This chapter deals with language features for dealing with **aggregates**, which are data that are structured according to some commonly useful organization, such as strings, arrays, and databases. Although many programming languages provide general-purpose facilities to structure data (such as records) and organize routines that manipulate the data (such as abstract data types), some structures are so important that languages deal with them specifically in order to make it easier to write clear and efficient programs. In this chapter, I concentrate on strings, arrays, databases, and mathematical formulas.

1 ◆ STRINGS

Most languages provide some facility for dealing with strings, that is, connected groups of characters. Some languages, however, specialize in string processing. This chapter will look at both elementary string operations and more complex control and data structures introduced in specialized string-processing languages.

1.1 Literals and Simple Operations

String literals are usually enclosed in double quotes ("). Some syntax is often provided to include unusual characters in string literals. For example, the C language allows an escape character to precede special forms, such as `\r` for a carriage return, `\t` for a tab, `\"` for a double quote, and `\023` for the character whose internal representation is octal 23. One nice escape sequence that doesn't exist in any language I know of skips to the next non-white text without including the white space in the string. I use `\c` to represent this special form, as in Figure 9.1.

On-line edition copyright © 1996 by Addison-Wesley Publishing Company. Permission is granted to print or photocopy this document for a fee of $0.02 per page, per copy, payable to Addison-Wesley Publishing Company. All other rights reserved.

Figure 9.1

```
StringVar := "this is a very long string that \c                    1
     I place on several lines, but it represents \c                  2
     a string without line breaks or gaps."                          3
```

Operations on strings are provided either by predefined procedures or by operators in the language. The simplest operations on strings, such as copying, equality testing, and lexical comparison, are often provided as overloaded meanings of `:=`, `=`, and `<`. Another simple operation is concatenation, often represented by the overloaded operator `+`. (SNOBOL represents concatenation by an empty space operator, which is quite confusing, particularly since the same invisible operator also represents pattern matching!) In addition, a few languages, such as ABC, provide operators for string repetition (`"ho" * 3` is `"hohoho"`), string length, and arcane operations such as finding the minimum character in a string.

Languages usually provide ways to convert other data types to strings. This facility is particularly important for output, which is often a long string computed from values of various types. Conversions to string are usually separate functions for each type to be converted, but C has a single function `sprintf` that can convert and concatenate any combination of basic types according to a format string, as in Figure 9.2.

Figure 9.2

```
IntVar := 23;                                                        1
sprintf(ResultString,                                                2
     "Give me %d number%s between %5g and 10%c.",                    3
     IntVar, if IntVar = 1 then "" else "s" end,                     4
     4.5, '0'");                                                     5
```

The format string in line 3 is copied to `ResultString`, but certain escapes prefixed by `%` cause later actual parameters to be converted and inserted into the string. The formats are specified by `%d` for integer, `%s` for string, `%g` for float, and `%c` for character. Formats can include width specifiers, as shown by `%5g`. This code places in `ResultString` the value

```
"Give me 23 numbers between  4.5 and 100."
```

A related and even simpler method is provided by Sal in the form of **edited strings** [Sturgill 89]. Figure 9.3 is the edited string equivalent of Figure 9.2.

Figure 9.3

```
IntVar := 23;                                                        1
ResultString :=                                                      2
     'Give me {IntVar} number\c                                      3
     {if IntVar = 1 then "" else "s" end} \c                         4
     between {4.5:5} and 10{'0'}.'                                   5
```

Expressions in braces are evaluated at runtime and formatted as appropriate to their type and according to any width specification given. Edited strings use a different set of delimiters from ordinary strings as a way to warn the compiler to inspect them for included expressions, which the compiler interprets to generate code. This code is executed when the edited string is first

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

evaluated; the result is an ordinary string that is not reevaluated later. Edited strings are more type-secure than the `sprintf` function, because there is no way to accidentally request that a value of some type be treated as a different type.

Languages often provide either a special syntax or a function call to extract substrings of a subject string based on position and length, as in Figure 9.4.

Figure 9.4

```
substr("A sample", 3, 4)
```

This string evaluates to `"samp"`, starting at the third position of `"A sample"` and continuing for 4 characters.

It is also common to provide for character or substring search. Search can be designed to return a Boolean to indicate success, the position of the character or substring if found (`0` otherwise), or a pointer to the character or substring if found (`nil` otherwise), as in Figure 9.5.

Figure 9.5

```
CharSearch("sample string", 's')                                    1
StringSearch("Target string", "get")                                2
```

The search in line 1 could return `true`, `1`, or a pointer to the entire string. The search in line 2 could return `true`, `4`, or a pointer to the substring `"get string"`. There might also be variants to conduct the search from right to left.

Slightly more complex than searching for characters is extracting data from a string while converting types; see Figure 9.6.

Figure 9.6

```
MyString := "4 and 4 make 8 in base 10"                             1
sscanf(MyString, "%d and %d make %g.", First, Second,               2
     Third); -- First := 4, Second := 4, Third := 8.0               3
```

Here the formats are used not to convert from numeric data to string data, but the reverse, to convert parts of the string into numeric data. The occurrences of `%d` in line 2 cause the substring `"4"` to be converted to the integer 4; the `%g` format converts `"8"` to the real `8.0`.

Another way to extract data from a string is to split it into fields based on some character. Several languages (for example, ABC, Perl, and Sal) provide a `split` procedure that takes a string and a set of characters considered to be field separators and a string array (passed by result) into which the given string is to be separated, as in Figure 9.7.

Figure 9.7

```
split("Veni, vidi, vici", ",", ResultArray)
```

This call would assign `"Veni"` into `ResultArray[0]`, `" vidi"` into `ResultArray[1]` (with the initial space), and `" vici"` into `ResultArray[2]`.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

1.2 Representation

Usually, programmers don't need to worry about how a language implementation represents strings. However, the representation can affect both the speed of computation and the way the program must manipulate strings. For example, C defines strings as consecutive characters terminated by a null (binary zero) character. This representation makes it slow to concatenate a string to the end of another (the implementation must find the end of the second string by a linear method) and does not allow nulls to be contained within strings. It encourages a programming style in which a variable points into the string and advances character by character until the terminating null is seen.

Alternative representations have some advantages. If the length of the string is encoded, perhaps in the first few bytes, then concatenation becomes faster, and strings may contain null characters. If strings are declared with a compile-time length, many operations become faster, and the compiler can keep track of the length of intermediate strings in complex expressions. However, some operations produce results whose length cannot be predicted. For example, a substring operation might take a variable length parameter. Therefore, languages in which strings are explicitly declared usually declare the maximum length that the string value might attain. This information determines storage requirements but does not dictate the length of particular values put into storage.

One attractive proposal is to omit a string-length code at the start of the storage area for a string, use a terminating null, but use the last byte of the storage area to indicate the distance back to the terminating null [Bron 89]. If the string just fits in the storage area, so that the terminating null is in the last place, the null looks like the number `0`, indicating zero distance to the terminating null. This representation makes it a bit harder for programs to build new strings directly, but a reasonable library of string-building operations can circumvent this problem, and programs may still scan through strings by using explicit pointers.

1.3 Pattern Matching

Sal, Awk, and Perl provide a match operator `~` that compares a target string to a regular expression. The result is Boolean, indicating success. A **regular expression** is a string, where most characters match themselves, but some characters and character combinations have special meanings. The following table lists some of these special meanings; Perl has an even richer set.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Regular expression	Matches
.	any character
\<	start of word
\>	end of word
\s	white space
\d	a digit (like [0-9])
\w	a word
^	the beginning of the target string
$	the end of the target string
[abc...]	any character in the set; ranges like: [3-7A-P]
[^abc...]	any character not in the set
r1 | r2	either r1 or r2 (alternation)
r*	zero or more r's
r+	one or more r's
r?	zero or one r's
r{3,5}	match 3, 4, or 5 r's
(r)	match r, call it a group
\2	string matched by the second group

Some of these expressions, like `.` and `\s`, match (and "use up") a single character. Others, like `\<` and `^`, do not use up any characters. For example, the `\<` expression matches the beginning of an alphanumeric region of the string; it is used to signal the start of a word. Grouping a subpattern allows you to refer to it later. Groups are numbered according to the left-to-right order of their opening parentheses. Figure 9.8 shows some examples of regular expressions.

Figure 9.8

```
"literal" -- matches "literal"                                    1
"l*iteral" -- matches "iteral", "literal", "llliteral" ...        2
"(l|b)(i|o)b\2" -- matches "libi", "lobo", "bibi", "bobo"         3
"[lb][io]b" -- matches "lib", "lob", "bib", "bob"                 4
```

The match operator can return the start and length of the matched substring via predeclared global variables or make them available through functions to be called after the match. If several matches are possible, one match is chosen. The usual rule is that `*` extends its match as far as possible and that the alternatives indicated by `|` are tried in the order given. In Perl, the search can be made insensitive to the case of the subject string, it can be made to start either at the beginning of the string or where the previous search left off, and it can be set not to extend the match as far as possible.

Slightly more sophisticated than matching a pattern is replacing the matched substring with new contents. The new contents can depend on parts of the matched patterns. Those parts are typically parenthesized groups, numbered in the order of their opening parentheses, as in Figure 9.9.

Figure 9.9

```
MyString := "here is a nice sample";                              1
Substitute(MyString, "(i(s) )", "wa\2");                          2
```

The `Substitute` procedure in line 2 assigns `"here was a nice sample"` to My-

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

String. The '\2' in line 2 fills in what the second group, (s), matched, namely, "s". Some languages provide sequences that can be placed in the third parameter of Substitute to indicate the part of the target string before the match, the entire matched part, and the part after the match, as in Figure 9.10.

Figure 9.10

```
MyString := "I think, therefore I am";                          1
Substitute(MyString, ",", " that \'\&\',");                     2
```

The expression \'\&\' in line 2 indicates the entire string, built up of the parts before, during, and after the match. The substitution changes MyString to "I think that I think, therefore I am, therefore I am".

1.4 Associative Arrays

Languages dealing with strings often provide a data type known as an **associative array**, which is indexed by strings instead of by integers or other scalar types. Associative arrays are usually implemented by hash tables. In some languages, like Sal and SNOBOL, the declaration of such an array indicates how large to make the hash table. If more elements are stored than the hash table size, access will become progressively slower but will still work. Other languages, like Perl, use extensible hashing and do not require any size declaration. ABC uses binary trees instead of hashing, so that a program can iterate through the array in key order. Other languages can only iterate in an implementation-dependent order.

Associative arrays are quite helpful in database applications. For example, to check for duplicates in a database with one field, say, StudentName, I could use the Boolean associative array Present of Figure 9.11.

Figure 9.11

```
variable                                                        1
     Present : array string of Boolean;                         2
     ThisEntry : string;                                        3

loop                                                            4
     ThisEntry := GetNextEntryOfDatabase();                     5
     if ThisEntry = "" then break end; -- exit loop             6
     if defined Present[ThisEntry] then -- found duplicate      7
          write("{ThisEntry} is a duplicate.");                 8
     end;                                                       9
     Present[ThisEntry] := true;                                10
end;                                                            11
```

In line 7, the **defined** operator indicates whether a value has been defined for the particular index value given; it returns a Boolean. The assignment in line 10 could just as easily use false; what counts is that some value is placed in Present[ThisEntry].

Associative arrays often come with a control structure for iterating over all index values that have been defined. Figure 9.12 continues the previous example.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 9.12

```
for Entry in Present do                                    1
     write(Entry);                                         2
end;                                                       3
```

1.5 Substrings as First-Class Values

Allowing substrings to be first-class values of a predeclared substring type has several advantages [Hansen 92]. Substring values can record not only their contents but also the identity of their base string. Dynamic allocation of space for substrings can be handled by the language at runtime.

Each value of the substring type contains a base string (perhaps implemented as a pointer) and the left and right positions in that string that delimit the substring. As with Icon, I understand positions to be between characters of the string.

The primitive operations on substrings can be simple and few. Here is a reasonable set of primitive operations:

- `start(x)`. Returns a substring with the same base as `x`, with both left and right set to left of `x`.
- `base(x)`. Returns a substring with the same base as `x`, left set before the first character of `x`, and right set to after the last character of `x`.
- `next(x)`. Returns a substring with the same base as `x`, left set to right of `x`, and right set one character after left if possible. Otherwise, right is set to the same position as left.
- `prev(x)`. Returns a substring with the same base as `x`, right set to left of `x`, and left set one character before right if possible. Otherwise, left is set to the same position as right.
- `extent(x,y)`. If `x` and `y` have different base strings, returns an empty substring of the empty base "". Otherwise, returns a substring with right set to the right of `y` and left set to either left of `x` or right of `y`, whichever is earlier in the base.
- `x = y`. The base strings of the two substrings are compared character by character between their left and right positions. The result is `true` if and only if the lengths are identical and the selected characters match exactly.
- `x + y`. Returns a substring containing a new base string that is the concatenation of the substrings `x` and `y`, and left and right at the beginning and end of that new base string.
- `x := y`. The old value of `x` is discarded; `x` acquires the same value as `y`, including the base string and the left and right positions.

Given these primitive operations, I can write a function that takes a substring representing a word terminated by blanks and returns a substring representing the next word, as in Figure 9.13 [Hansen 92].

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 9.13

```
function NextWord(value aWord : substring) : substring;          1
begin                                                            2
     loop -- skip to end of word                                 3
          aWord := next(aWord);                                  4
          if aWord ≠ " " then break end;                         5
     end;                                                        6
     while next(aWord) ≠ "" and next(aWord) ≠ " " do             7
          aWord := extent(aWord, next(aWord));                   8
     end;                                                        9
     return aWord;                                               10
end; -- NextWord                                                 11
```

The primitive substring operations can be used to build slightly more sophisticated operations, such as `rest`, which returns all but the first character of its substring parameter, and `last`, which returns just the last character of its substring parameter. They can also be used to build Icon's matching procedures.

1.6 SNOBOL

SNOBOL was developed by Ralph E. Griswold and others at Bell Telephone Laboratories around 1965 [Griswold 71]. It has a strange syntax, partially because it was developed before Algol-like syntax became popular. Spaces act as both the concatenation and the match operators. The only statement form includes pattern match, replacement, and success and failure **goto**s. To avoid confusion, I translate all the SNOBOL examples into an Ada-like syntax, using **match** and **replace** operators. SNOBOL uses dynamic typing and dynamic scope rules; its primitive data types are strings, integers, and reals. The structured types include patterns (distinct from strings), nonhomogeneous arrays, and associative arrays.

Variables are not declared; all conceivable strings (even the empty string) name variables. Initially, all variables have the value "". In a sense, therefore, all string values point to other strings, as in Figure 9.14.

Figure 9.14

```
somewhere := "over";                                             1
over := "the";                                                   2
the := "rainbow";                                                3
write(somewhere^^); -- writes "rainbow"                          4
```

SNOBOL is homoiconic, after a fashion. A program is a string, and it is possible at runtime to compile a string and to branch to a label in it. However, this facility is much less attractive than LISP's equal treatment of program and data structure. SNOBOL has not been heavily used for artificial intelligence programming.

SNOBOL patterns are like regular expressions, but more powerful. They are structured values built recursively. The simplest patterns are string literals and string-valued expressions, which match themselves. More complex patterns are formed by sequencing (somewhat like **and**), alternation (somewhat like **or**), and by invoking pattern-returning predeclared functions. Patterns are matched by a backtracking algorithm, trying earlier alternatives first. Backtracking in pattern matching is very similar to backtracking in

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

logic programs (see Chapter 8). Consider Figure 9.15.

Figure 9.15

```
aString := "The boy stood on the burning deck, \c                    1
     Eating peanuts by the peck.";                                   2
aPattern := ("ing" | "the") & " " & ("deck" | "peck");               3
aString match aPattern;                                             4
```

The pattern in line 3 includes alternation, represented by `|`, and sequencing, represented by `&`. The `|` operator indicates that if the pattern on its left fails to match, the pattern on its right should be tried. The `&` operator indicates that if the pattern on its left succeeds, the pattern on its right should then be matched at the position following the match of the pattern on the left. If the pattern on the right fails, the pattern on the left is retried. Line 4 would succeed, matching "`ing deck`". If forced to backtrack, it would match "`the peck`".

The predeclared pattern-returning functions are as follows:

Pattern	Matches
`len(4)`	any string of 4 characters
`tab(5)`	to position 5 of the string
`rtab(6)`	to position 6 from the end of the string
`pos(7)`	succeeds if at position 7; matches empty string
`rpos(7)`	succeeds if at position 7 from right; matches empty string
`any("abc")`	any character in the set
`notany("abc")`	any character not in the set
`span("abc")`	until a character not in the set
`break("abc")`	until a character in the set
`rem`	the remainder of the string
`arb`	0 chars, on reevaluation any 1 char, then 2, and so on
`bal`	like `arb`, but not matching unbalanced parentheses

Special patterns control backtracking. The pattern **`fence`** succeeds, but backtracking refuses to reevaluate it. It is equivalent to Prolog's **`cut`** operator, except that it does not prevent alternatives elsewhere in the pattern from being tried. The **`succeed`** pattern succeeds the first time and all succeeding times; consider Figure 9.16.

Figure 9.16

```
"a string" match (succeed & "p")
```

This match will never terminate, because **`succeed`** will continue to retry, even though "`p`" keeps failing. A related pattern is **`fail`**, which fails each time it is attempted. It is used to force subsequent matches of the previous part of the pattern, usually for the side effects that matching can produce. Finally, **`abort`** causes the match attempt to terminate entirely with failure.

SNOBOL programmers often employ patterns for their side effects. The matched substring may be replaced by a new string, as in Figure 9.17.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 9.17

```
far := "away";                                    1
far match "y" replace "ke";                       2
```

Line 2 will assign "awake" into far. The part of the string matched by a subpattern can be immediately assigned into a variable, as in Figure 9.18.

Figure 9.18

```
there := "dream";                                 1
pat := (len(3) =: bluebird);                      2
there match pat;                                  3
```

The pattern has a side effect, to assign into variable bluebird the results of matching the subpattern len(3). The **match** in line 3 will succeed and will assign "dre" to bluebird. I have used =: to denote the **immediate assignment** operator. The side effect of assignment takes place as soon as the immediate assignment operator is encountered during pattern matching. I can use immediate assignment to construct a pattern that will match any doubled string, as in Figure 9.19.

Figure 9.19

```
pat := pos(0) & (arb =: firstpart) & (delay firstpart) &      1
     rpos(0);                                     2
"abab" match pat; -- succeeds                     3
```

The four components of the pattern in line 1 are sequenced together. The pos(0) and rpos(0) components force the rest of the pattern to apply to the entire subject string. The predefined pattern arb matches any length string, starting with the empty string. Whatever it matches is immediately assigned to firstpart. The pattern then looks for firstpart itself, that is, a repetition of the first part. The unary **delay** operator forces lazy evaluation of its argument. Otherwise, the value of firstpart at the time the pattern is constructed would be embedded in the pattern instead of its value at the time the pattern is evaluated during matching. When the pattern is applied in line 2, arb first matches "", so **delay** firstpart also matches "". But rpos(0) fails, so matching backs up. The pattern **delay** firstpart fails to find an alternative, but arb finds the alternative "a". This time, **delay** firstpart fails. The next alternative for arb is "ab", and this time the entire match succeeds.

In addition to immediate assignment, SNOBOL also provides **conditional assignment**, placing the value of a matched substring in a variable only if the match completely succeeds. Conditional assignment tends to be more efficient than immediate assignment, since it can avoid multiple assignments as the pattern match backtracks, but it can't be used in the double-word example. Finally, the **position assignment** operator @ assigns the position in the subject string (that is, a number such as 6) to a variable during matching.

Programmers often use immediate and conditional assignment to assign values to the pseudovariable output. Every assignment to output causes the value to be output from the program. Similarly, every evaluation of input reads in a value from the program.

SNOBOL allows an arbitrary procedure call to be inserted in a pattern. The value returned by the procedure is treated as part of the pattern being matched. (String values are coerced to patterns for this purpose.) Usually, such a call is prefixed by the **delay** operator to postpone the evaluation of the

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

actual parameters and the invocation of the procedure until match time. If the procedure fails, then that part of the pattern match fails, and backtracking takes over. Information resulting from the match so far can be passed to the procedure via immediate assignment to global variables or to local variables passed as actual parameters.

1.7 Icon

Icon was developed by Ralph E. Griswold, one of the developers of SNOBOL, in the late 1970s as a result of his dissatisfaction with how SNOBOL's patterns fit into the language [Griswold 80]. It retains the virtues of SNOBOL's pattern matching without a pattern data type. It is an expression-oriented language, with each evaluation resulting in either a value (counted as a success) or failure. Instead of using Boolean values, conditionals base their actions on the success or failure of evaluating their conditions.

The first novel idea in Icon is the **scan** statement. (I call it a statement, even though all constructs in Icon are actually expressions, because it is usually not used for its value.) This statement introduces a name scope that creates a new binding for two predeclared variables, subject and pos, which specify the current string being matched and the current position within the string. Consider Figure 9.20 (I take liberties with actual Icon syntax to keep my examples consistent).

Figure 9.20

```
scan "peristalsis" using                                    1
     write("[" + move(4) + "]")                             2
end;                                                        3
```

This program prints "[peri]". The **scan** in line 1 maps subject to "peristalsis" and sets pos initially to 1. The body of **scan** is in line 2; it implicitly uses both subject and pos (modifying the latter). The predeclared procedure move causes the position to be incremented, if subject is long enough, and if it succeeds, it returns the substring of subject over which it has advanced. The + operator is string concatenation. After the body, both subject and pos revert to whatever values they had before. Figure 9.21 shows a more complex nested example.

Figure 9.21

```
scan MyString using                                         1
     loop -- each iteration deals with one word             2
          scan tab(upto(" ")) using                         3
               if upto("-") then -- word has a hyphen       4
                    write(subject);                         5
               end;                                         6
          end; -- scan tab(upto(" "))                       7
          move(1); -- past " "                              8
     end; -- loop                                           9
end; -- scan MyString                                       10
```

This program prints out all space-delimited words in MyString that contain a hyphen. The outer **scan** (lines 1–10) contains a loop that repeatedly advances pos to a space, scans the intervening word (lines 3–7), and then moves past the space (line 8). The predefined function upto (lines 3 and 4) returns the

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

position of the first occurrence of any character in its actual parameter. If there is no such occurrence, it fails, and this failure is tested by a conditional (line 4). The function tab (line 3) moves pos to the value of its actual parameter and returns the substring of subject that it has moved over (in either direction). The expression in line 3 is interpreted in the outer scope; that is, it moves the cursor in MyString, and the move in line 8 moves the cursor again. The inner scope, lines 4–6, has its own subject and pos. Even if it modified pos (it doesn't), that modification would not be seen by the outer scope.

The pattern-returning functions of SNOBOL are replaced in Icon by a small set of predeclared matching procedures, which return either positions or matched strings if they succeed, and which can have the side effect of modifying pos. These are the procedures:

Procedure	Returns	Side effect
tab(n)	string between pos and n	pos := n
move(n)	string between pos and pos + n	pos := pos + n
upto(s)	position of next character in s	none
many(s)	position after 0, 1, . . . characters in s	none
any(s)	pos + 1 if current character in s	none
find(s)	position before first occurrence of s	none
match(s)	position after s starting at pos	none
bal()	position of end of balanced string starting at pos	none

The first procedures, tab and move, are the only ones that modify pos. Instead of numbering character positions, Icon indexes strings between characters, starting with 1 before the first character of a string. This convention makes it unnecessary to say such things as "up to and including position 4." Each intercharacter position has an alternative index, which is 0 at the end of the string and increasingly negative toward the front of the string. So tab(0) moves to the end of the string, and tab(-3) moves before the character 3 before the end. If tab or move would exceed the limits of the string, they fail and have no side effect.

The remaining procedures examine subject and return a position that can be given to tab or move. For example, to move past "ThisString", I could write the expression in Figure 9.22.

Figure 9.22

```
tab(match("ThisString"))
```

Icon lets the programmer introduce new matching procedures. The currently active pos and subject are automatically inherited by procedures, since Icon uses dynamic scope rules. Procedures may directly modify pos, or they may indirectly modify it by invoking other matching procedures, such as the predefined ones. Usually, though, they are designed only to return a position, and the invoker may then use tab to modify pos. Figure 9.23 shows a procedure MatchDouble that looks for the given string twice in succession:

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 9.23

```
procedure MatchDouble(Given) : integer;                    1
     return match(Given + Given);                          2
end;                                                       3
```

The **return** statement in line 2 returns failure if its expression fails. A programmer may also explicitly return failure by a **fail** statement.

The second novel idea in Icon is that each expression is, either implicitly or explicitly, an iterator in the CLU sense, as discussed in Chapter 2. Backtracking can require that an expression be reevaluated, and it may produce a different result the next time.

Some matching procedures, such as match and pos, fail if reevaluated. The reason is that if the first success is not good enough for whatever invoked it, it wasn't the fault of the procedure, which has no better result to offer. Other matching procedures try to find additional answers if reevaluated. For example, upto("a") applied to "banana" at position 1 will first return 2, and on successive evaluations will return 4, 6, and then failure. Likewise, find and bal locate matches further and further from the original position.

Backtracking causes the previous value of pos to be restored before reevaluation. Reevaluation of a procedure invocation first tries new answers from the procedure without changing the actual parameter and then tries reevaluating the actual parameter. For example, tab(upto("a")) applied to "banana" can be reevaluated after it has succeeded in moving pos to 2. Since tab fails on reevaluation, its parameter upto("a") is reevaluated. This reevaluation is in the context before tab had advanced pos; that is, pos is first restored to 1. Now upto("a") returns 4, so tab will set pos to 4.

The real novelty comes from the fact that the programmer can explicitly build iterator expressions without using predefined matching procedures. Such expressions can be built with the **alternation** operator | . For example, 4 | 3 is an iterator expression with values 4, 3, then failure. Iterator expressions can be used anywhere an expression is expected, such as an actual parameter. When first evaluated, tab(4 | 3) moves pos to 4. If it is reevaluated, it moves pos to 3 instead. Further evaluations lead to failure.

The sequence operator & also builds iterator expressions, as in Figure 9.24.

Figure 9.24

```
scan "malarky" using                                       1
     write(tab(upto("a")) & match("ark")); -- outputs 7    2
end;                                                       3
```

In line 2, upto("a") returns 2, tab advances pos to 2, and match("ark") fails. The sequence operator causes tab to reevaluate, which fails, causing upto("a") to reevaluate, returning 4. Now tab advances pos to 4, and match("ark") succeeds, returning 7. The result of the sequence operator is its second operand, so write outputs 7. If the sequence operator were replaced by ; , match("ark") would fail once, and write would not be called at all.

Iterator expressions are useful in many surprising contexts, such as in conditional and iterative statements; consider Figure 9.25.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 9.25

```
if (ThisVar | ThatVar) = (5 | 2 | 10) then ...                    1
while LowBound < (ThisVar & ThatVar) do ...                       2
```

In line 1, if ThisVar = 4 and ThatVar = 5, reevaluation stops after the second alternative of the first clause and the first alternative of the second clause; ThatVar is not compared against 2 and 10. Line 2 shows a nice shorthand for LowBound < ThisVar **and** LowBound < ThatVar.

Backtrack can be invoked directly by an **every** statement, as in Figure 9.26.

Figure 9.26

```
scan "malarky" using                                              1
     every place := upto("a") do                                  2
          write(place); -- 2, 4                                   3
     end;                                                         4
end;                                                              5
```

This program outputs both 2 and 4. The **every** statement in lines 2–4 reevaluates place := upto("a") until it fails; for each successful evaluation, line 3 is executed.

Iterator procedures look just like any other procedure, except that they use **yield** to return a value. Figure 9.27 converts the MatchDouble procedure of Figure 9.23 (page 281) to an iterator that will return the position after any double instance of its parameter.

Figure 9.27

```
procedure MatchDouble(Given : string) : integer;                  1
     variable place : integer;                                    2
     every place := find(Given + Given) do                        3
          yield place + 2*length(Given)                           4
     end;                                                         5
end;                                                              6

-- sample use                                                     7
scan "committee meets three times" using                          8
     variable here : integer;                                     9
     every here := MatchDouble("e") do                            10
          write(here); -- 10, 14, 22                              11
     end;                                                         12
end;                                                              13
```

Iterator procedures can be used to parse using a BNF grammar. For example, the grammar of balanced parentheses is shown in Figure 9.28.

Figure 9.28

```
Bal ::= ε | "(" Bal ")" Bal
```

An iterator procedure that finds longer and longer balanced parenthesis strings appears in Figure 9.29.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 9.29

```
procedure Bal() : integer;                                    1
     every                                                    2
          match("") | (                                       3
               tab(match("(")) & tab(Bal()) &                  4
               tab(match(")")) & tab(Bal())                    5
          )                                                    6
     do                                                       7
          yield pos;                                           8
     end;                                                     9
end;                                                          10

-- sample use                                                 11
scan "()(())(" using                                          12
     variable here : integer;                                 13
     every here := Bal() do                                   14
          write(here); -- 1, 3, 7                             15
     end;                                                     16
end;                                                          17
```

1.8 Homoiconic Use of Strings: Tcl

Several syntax rules in Tcl interact to make it homoiconic. Lists are represented as strings; the individual elements are delimited by white space. Every string names a variable. The R-value of a variable is denoted by `$` before the string that represents the variable. (This rule makes Tcl programs error-prone, because it is so easy to forget the `$`.) Strings need not be delimited by quotes unless they have embedded spaces. There are quotes (`{` and `}`) that prevent any evaluation within a string, quotes (`"`) that allow evaluation, and quotes (`[` and `]`) that force the string to be evaluated. Evaluating a string means treating it as a series of commands delimited by end-of-line characters or semicolons. Each command is the name of a procedure (many are predeclared; I will show them in **`bold monospace`**) followed by parameters. The whole program is a string to be evaluated. Figure 9.30 shows a simple Tcl example.

Figure 9.30

```
set a 4 -- a := 4                                             1
set b [expr $a + 5] -- b := 9                                 2
while {$b > 0} {                                              3
     puts "b is now $b"                                       4
     set b [expr $b - 2]                                      5
}                                                             6
```

This program prints `b is now 9` and then four more similar outputs. Line 1 is the assignment statement. It takes the name, not the R-value, of the variable to be assigned. Line 2 shows the quotes that force evaluation: `[` and `]`. The **`expr`** command evaluates any number of parameters as an arithmetic expression. It returns the value of that expression. Line 3 introduces the quotes that prevent evaluation: `{` and `}`. The **`while`** command takes two unevaluated strings, the first representing a conditional and the second representing the body of the loop. It repeatedly invokes **`expr`** on the first parameter, and if the result is true, it evaluates the second parameter, thereby

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

executing the body. The body contains end-of-line characters, allowing the parser to separate it into individual statements. Line 4 shows the last kind of quotes, which can build a string containing spaces, but which do not prevent evaluation of such constructs as $b.

To see how Tcl is homoiconic, consider Figure 9.31, a less readable version of the same program.

Figure 9.31

```
set a 4 -- a := 4                                                    1
set rhs {expr $a +} -- rhs := "expr $a +"                            2
set rhs [append rhs 5] -- rhs := "expr $a + 5"                       3
set b [eval $rhs] -- b := 9                                          4
set cond {$b > 0} -- cond := "$b > 0"                                5
set body {                                                           6
     puts "b is now $b"                                              7
     set b [expr $b - 2]                                             8
}                                                                    9
while $cond $body                                                    10
```

The condition and the body of the **while** loop in line 10 are the result of previous computations. Even commands can be computed, as in Figure 9.32.

Figure 9.32

```
set a ile -- a := "ile"                                              1
wh$a {$b > 0} {set b [expr $b - 2]}                                  2
```

Line 2 is actually a **while** command, because the first word evaluates to while.

2 ◆ ARRAYS: APL

Arrays are especially important in mathematical computation. One of the principal advances in FORTRAN 90 over earlier versions of FORTRAN is its ability to manipulate arrays without dealing with the individual array elements. However, the best example of an array language is not FORTRAN, but APL. The APL language was invented by Kenneth E. Iverson in the early 1960s and has had a small but devoted following ever since. It could be considered a single-minded language: All computation is cast in the mold of array manipulation. Its practitioners point with pride at the conciseness of their programs; detractors point with scorn at the unreadability of the same programs. APL has long suffered from the fact that most of its operators are not normal ASCII symbols, so ordinary keyboards are not adequate for representing APL programs. Dialects such as J and APL/11 use several ASCII characters together to represent the unusual symbols. My examples expand unusual symbols into keywords to help you read them.

APL programs must be studied; they cannot simply be read. Not only does APL have an unusual character set, but it lacks control structures such as **while** and conditionals.

APL's greatest strength is its ability to handle arrays of any dimension with the same operators that apply to scalars (which are zero-dimensional arrays). The meaning is to apply the operator pointwise to each member of the array. The resulting uniformity, along with the wealth of arithmetic opera-

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

tors, makes it quite a powerful language. Another contributor to uniformity is that Booleans are represented (as in C) as numeric values: 0 means `false` and 1 means `true`. Arrays of Booleans can therefore be manipulated by the same means as arrays of numbers. Similarly, strings are treated as arrays of characters and can also be handled identically to numeric arrays.

If an operator requires both operands to have the same dimension, it is often valid to apply that operator to operands of different dimension. For example, `x + y` is the pointwise addition of elements of `x` with elements of `y`. Suppose that `y` is a matrix (that is, two-dimensional) with bounds 5 and 6, and that `x` is a scalar (zero-dimensional) with value 4. Then `x` will be coerced to two dimensions to conform to `y`, and each cell of the coerced matrix will have value 4. This kind of coercion is called **spreading**. The value `x` can be spread to conform to `y` only if the bounds of the dimensions of `x` match the bounds of the initial dimensions of `y`. In this example, `x` has no dimensions, so the condition is trivially met. Most APL implementations only allow one-dimensional quantities to be spread.

2.1 Operators and Meta-operators

APL is generally interpreted, not compiled. All operators are right-associative and have the same precedence. Most operator symbols can be used either as unary or as binary operators, often with different meanings. To keep things clear, I use different keywords for the two meanings. Besides ordinary operators such as `+`, APL has many unusual operators, including the following:

Operator	Meaning
`x min y`	min(x,y) -- lesser value
`floor x`	floor(x) -- greatest integer ≤ x
`ceil x`	ceiling(x) -- least integer ≥ x
`recip x`	1/x -- reciprocal
`sign x`	abs(x) / x -- sign of x
`abs x`	abs(x) -- absolute value
`x max y`	max(x,y) -- greater value
`exp x`	exp(x) -- e to power x
`x power y`	x ^y -- x to power y
`x log y`	logarithm (base x) of y
`ln x`	logarithm (base e) of x
`x comb y`	C(y,x) -- number of combinations of y taken x at a time
`fact x`	factorial(x) -- x can be fractional
`x deal y`	x integers picked randomly (no replacement) from 1...y
`rand x`	random integer from 1..ceiling(x)
`x layout y`	array with dimensions x and initial value y
`fill x`	one-dimensional array with initial values 1...x
`shape x`	array of bounds of x
`x drop y`	remove first x elements of y
`x take y`	keep only first x elements of y
`transpose x`	reverse the order of dimensions of x
`x member y`	0 or 1, depending on whether x is found in y
`x cat y`	x concatenated with y (spread if necessary)

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

ravel x	array x reduced to one dimension (row-major)
x **rotate** y	array y left-rotated in first dimension by x places
x **matdiv** y	x / y, where both are matrices
matinv x	inverse(x), where x is a matrix
x **compress** y	only members of y in positions where x is true

If you call an operator a verb, then APL provides not only many verbs but also a few adverbs that modify verbs. You might call adverbs **meta-operators**, because they convert operators to new operators. Here are some meta-operators, where v and w are the operators on which they act.

Adverb	Meaning
x **outer** v y	outer product with operator v on x and y
x v **inner** w y	inner product with operators v and w on x and y
v **accumulate** x	apply operator v to one-dimensional array x repeatedly
v **scan** x	accumulate, generating all intermediate results
x v **rank** n y	operator v applied to n-dim cells of x and y
x v **birank** n m y	operator v applied to n-dim cells of x and m-dim cells of y
n **power** v x	operator v applied n times to x.

The operators v and w can be any binary operators, including programmer-defined procedures. This ability to create new operators out of old ones is quite powerful indeed. The **power** operator is equivalent in purpose to power loops, described in Chapter 2.

Figure 9.33 presents some examples to help clarify this welter of operators.

Figure 9.33

```
in:  3 4 5 -- one-dimensional array                          1
out: 3 4 5                                                   2

in:  a := 3 4 5                                              3
     recip a -- applies pointwise to each element            4
out: .333333333 .25 .2                                       5

in:  3 + a -- 3 is spread to same dimension as a             6
out: 6 7 8                                                   7

in:  + accumulate a -- like 3 + 4 + 5                        8
out: 12                                                      9

in:  - accumulate a -- like 3 - (4 - 5)                      10
out: 4                                                       11

in:  - scan a -- 3, 3-4, 3-(4-5)                             12
out: 3 -1 4                                                  13
```

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

```
in:  max accumulate a                                          14
out: 5                                                         15

in:  * accumulate recip a -- .333333333 * .25 * .2             16
out: .0166666667                                               17

in:  a=a -- pointwise comparison                               18
out: 1 1 1                                                     19

in:  ≠ accumulate a=a -- determine parity                      20
out: 1                                                         21

in:  fill 4                                                    22
out: 1 2 3 4                                                   23

in:  recip fill 4                                              24
out: 1 .5 .333333333 .25                                       25

in:  (2 3) layout fill 6                                       26
out: 1 2 3                                                     27
     4 5 6                                                     28

in:  a := (2 3) layout fill 6                                  29
     a[1,1] := 9 -- indices start at 1                         30
     a[2,] := 8 -- entire row; 8 is spread                     31
     a[,2] := 7 -- entire column; 7 is spread                  32
     a                                                         33
out: 9 7 3                                                     34
     8 7 8                                                     35

in:  (2 3) layout (5 6) -- last parens not needed              36
out: 5 6 5                                                     37
     6 5 6                                                     38

in:  + accumulate (2 3) layout (5 6)                           39
out: 16 17                                                     40

in:  + scan (2 3) layout (5 6)                                 41
out: 5 11 16                                                   42
     6 11 17                                                   43

in:  1 rotate (3 2) layout fill 6                              44
out: 3 4                                                       45
     5 6                                                       46
     1 2                                                       47

in:  (fill 4) + inner * (fill 4)                               48
     -- sum of products; last parens not needed               49
out: 30                                                        50

in:  (fill 2) + inner * ((2 3) layout fill 6)                  51
          -- sum of products                                   52
out: 9 12 15                                                   53
```

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

```
in:  (fill 2) * inner + ((2 3) layout fill 6)                    54
          -- product of sums                                      55
out: 12 21 32                                                    56

in:  (fill 2) outer + (fill 2)                                   57
out: 2 3                                                         58
     3 4                                                         59

in:  (fill 2) outer * (fill 2)                                   60
out: 1 2                                                         61
     2 4                                                         62

in:  (1 2 3) cat (4 5 6)                                         63
out: (1 2 3 4 5 6)                                               64

in:  (1 2 3) cat rank 0 (4 5 6)                                  65
out: 1 4                                                         66
     2 5                                                         67
     3 6                                                         68
```

As you can see, APL allows a great many usual and unusual manipulations to be performed readily. The computations lend themselves to vector-processing hardware on modern supercomputers.

Although APL has no structured control structures, it does have **goto**, and the label can be computed, as in Figure 9.34.

Figure 9.34

```
goto ((a > 0) cat (a < 0) cat (a=0)) compress                    1
     (positive cat negative cat zero)                            2
```

Line 2 builds an array of labels (I ignore how labels are declared). Line 1 compresses that array to one element based on a Boolean array only one of whose elements can be `true`. It then executes a **goto** to the selected label.

Figure 9.35 shows how to generate the first *n* Fibonacci numbers.

Figure 9.35

```
(n - 2) power                                                    1
     (right cat + accumulate -2 take right) -- OneStep           2
     1 1                                                         3
```

The **power** meta-operator replicates the anonymous operator given in line 2 (let me call it `OneStep`) `n-2` times and then applies the resulting operator to the array `1 1`. `OneStep` uses the predeclared identifier `right` to refer to its right-hand operand, which is an initial Fibonacci string. Since it has no occurrence of `left`, `OneStep` is unary. `OneStep` takes the *last* two elements of the operand, since the left argument to **take** is a negative number. These last two elements are summed by the accumulation and are then concatenated to the previous sequence.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

2.2 An APL Evaluator

One of the most delightful things about APL is that it lends itself to lazy evaluation. For example, **transpose** need not actually create a new array and fill it with data; it needs only to wait until one of its values is required. It can then convert the indices of the desired access into the nontransposed indices and fetch the value from its operand. Likewise, the **fill** operator need not actually build an array; it can easily return values when they are actually required. Although lazy evaluation will generally not be faster than full evaluation, it can avoid allocating large amounts of space.

A lazy evaluator can be written for APL in an object-oriented language. In Smalltalk nomenclature, the class Expression has instance variables dimension, bounds, and values. For example, (3 4) **layout** 4 can be represented by an object in which dimension = 2 and bounds = (3 4). The instance variable values caches the values that have already been computed, so they do not need to be computed again. The Expression class has a method **inRange:** that reports whether a given index expression is valid for the dimensions and bounds given. It also provides methods **store:at:** and **retrieve:at:** for caching computed values in values, a method **write** for displaying all values, and methods **dimension** and **bounds** to report these instance variables.

The Expression class has subclasses for every operator. Each subclass has methods for initialization (to set the dimension and bounds) and for access at any index. For example, Fill sets dimension = 1. It can compute the value at any valid index without needing to store any array. Subclasses like Matinv that wish to cache computed values may do so via **store:at:**. The Expression class has methods for creating and initializing an instance of each subclass. One final subclass of Expression is Spread, which is used to accomplish coercion to a higher dimension. It can be called explicitly, but it will also be called implicitly by operators such as Plus when necessary.

Some of the examples above could be cast as shown in Figure 9.36 into Smalltalk.

Figure 9.36

```
APL: fill 5                                                    1
OOP: Expression fill: 5                                        2

APL: ≠ accumulate a=a                                          3
OOP: Expression accumulate: NotEqual of:                       4
          (Expression equal: a and: a)                         5

APL: (fill 4) + inner * (fill 4)                               6
OOP: Expression inner: Plus with: Times of:                    7
          (Expression fill: 4) and: (Expression fill: 4)       8

APL: (2 3) layout fill 6                                       9
OOP: Expression layout: #(2 3) with:                           10
          (Expression fill: 6)                                 11
```

In Line 2, the **fill:** method in Expression returns an instance of the Fill subclass, suitably initialized. I have omitted an invocation to **write** that would display all the values of this object. In lines 4–5, the **accumulate:of:** method of Expression creates an instance of the Accumulate subclass and

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

gives it both an operator, represented as the class `NotEqual`, and an expression to manipulate (all of line 5). If it needs to make calculations, it can instantiate `NotEqual` as many times as needed and initialize those instances to the appropriate values. Array literals such as `#(2 3)` (line 10) could be coerced to the appropriate constant `Expression`, or I could require that they be explicitly converted by saying `Expression constant: #(2 3)`.

2.3 Incremental Evaluation

In some applications, the same program is executed repeatedly on slightly different inputs. For example, spreadsheet programs are often reevaluated with slightly different data. Functional programming languages have been designed that can quickly evaluate expressions given new data expressed as a modification of previous data [Yellin 91].

I want to show you how this idea can be embedded in an APL interpreter. To keep the discussion simple, I do not use a lazy interpreter, and I assume that the program is a single function with no internal variables. Given an old value and a new value, a **delta** represents how to change the old value to the new value. Of course, by value I mean an array of some shape. The delta and the old value together are enough to completely specify the new value.

Every operator instance records the most recent value it has produced. It provides that value to its caller as a delta. The ultimate caller is typically the outer-level `write` routine, which uses the delta it receives to display the value of the program. (It might even display the delta, if the user is interested in that representation instead of the fully expanded result.) The first time the program is run, the deltas show the difference between the void value (not even a zero-dimensional array!) and the initial value.

In order for this scheme to be efficient, incremental computation should usually not be more expensive than computing from scratch. If we are lucky, incremental computation is very inexpensive. An occasional inefficient recomputation is perfectly acceptable, though.

Achieving efficiency has two parts. First, the format for the deltas should not be longer than the new value. If a value has changed in major ways, it is better just to provide the new value outright. For APL arrays, a delta might indicate dimensions to delete, indices within a dimension to delete, particular values to change, and new indices within a dimension to add (with their values). For example, the delta from `1 3 4 5 7` to `1 2 4 5` might be represented as "change at index 2 to value 2, delete index 5."

Second, each operator and meta-operator should be implemented to take advantage of deltas. For example, the `+` operator generates an output delta that only includes indices where the input deltas indicate a change. The **`accumulate`** meta-operator could make use of an inverse to the operator it is given, if one exists, in order to remove the effects of any deleted array elements before adding the effects of inserted elements.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

3 ◆ DATABASE LANGUAGES

Databases are much more varied in structure than strings or arrays. The range of languages designed for databases is also quite wide. Database languages tend to look like ordinary algebraic languages and are often Algol-based. They integrate database operations by providing additional data types and control constructs. Typically, programmers need to keep two "current locations" in mind: the current point of execution of the program, and the current record of a database. In some languages, it is also necessary to keep the current relation in mind.

3.1 Data Types

There are several ways to represent data in a database, known as hierarchical, network, and relational. I concentrate on relational databases, in which information is stored in **relations**, which are persistent homogeneous arrays of records.

My examples are taken from dBASE [Simpson 87], Sal [Sturgill 89], and a higher-level language, SQL. My examples will be based on the relations shown in Figure 9.37.

Figure 9.37

```
People : relation                                    1
     FirstName, LastName : string;                   2
     BirthYear : integer;                            3
end;                                                 4

Events : relation                                    5
     Place, What : string;                           6
     EventYear : integer;                            7
end;                                                 8
```

That is, People and Events are homogeneous persistent arrays of records with the fields as shown. I have not limited the length of the string fields (dBASE requires declaring the exact length; Sal does not, but does allow patterns that restrict valid values) nor the range of the integer fields (dBASE requires specifying the number of characters in a string version of the field; Sal allows explicit range specification). The data specifications, known as **schemata**, are usually stored in files, as are the relations themselves. Schemata are built either interactively (dBASE) or by a specification file (Sal).

In dBASE, a program that uses a relation opens it for use, at which time the field names become defined. dBASE is dynamic-typed. Runtime functions are available to determine the types of fields. In Sal, a program must read the relation into a local relation variable before using it and must specify which fields are to be read. Runtime type checking verifies that the specified fields actually exist and are consistent with the uses to which they are put.

Both dBASE and Sal allow the programmer to restrict attention to those records in a relation for which some Boolean expression holds. In dBASE, there are two techniques for restriction. First, a **filter** statement causes records to be invisible, as in Figure 9.38.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 9.38

```
filter BirthYear < 1990 and LastName ≠ FirstName;
```

Until filtering is turned off, accesses to the currently open relation will not see any record for which field BirthYear ≥ 1990 or for which LastName = FirstName. This statement causes a runtime error if the currently open database does not have fields with the given names or if there is a type mismatch (for example, if BirthYear is not compatible with integer).

Second, control constructs that iterate through a relation can explicitly avoid certain records, as I describe shortly. In Sal, the statement to copy an external relation into an internal variable has an optional **where** clause to select appropriate records only. The advantage of Sal's approach is that the same relation can be read into multiple variables, possibly with different restrictions, after which each can be independently accessed. In dBASE, it is not possible to have two filters on the same relation, nor to have the same relation open multiple times. It is easy in Sal, but quite awkward in dBASE, to generate a list of all first names combined with all last names. The advantage of dBASE's approach is that entire relations do not need to be read into memory before access may begin. Large relations are not usable in Sal. Of course, Sal could be implemented to evaluate relation variables in a lazy fashion or to represent them on external store altogether.

Both languages allow the programmer to construct Boolean expressions involving the fields of a relation. In addition to arithmetic and string comparison, both have pattern matching. In dBASE, pattern matching is restricted to determining if one string expression is contained within another. dBASE also has an inexact string-comparison mode in which strings are considered equal if the first is a prefix of the second. Sal has a regular-expression pattern matcher.

In dBASE, multiple orders can be imposed on the records of a single relation. They include natural order (the order in which records have been added to the relation) and sorting (either increasing or decreasing) on any field or expression based on fields. These orders are built under program control, are given names, and persist after the program finishes, as shown in Figure 9.39.

Figure 9.39

```
open People; -- make the relation available and current          1
order BirthOrder; -- increasing BirthYear                        2
seek 1950; -- move to the first record matching expression       3
makeorder NameOrder := LastName + " " + FirstName;               4
order NameOrder; -- use the order                                5
seek "Newman Alfred"                                             6
```

In line 1, People is opened for use. Line 2 establishes which order is to be used. BirthOrder must already be part of the persistent representation of the relation. Line 3 moves the current-record mark to the first record for which 1950 is the value under the current order. The programmer needs to remember the expression that defines BirthOrder, since it is not given in the program. I am assuming it is simply the BirthYear field and is of type integer. Line 4 shows how a new order can be added to the relation and given a name. I use + for string concatenation. The success of the **seek** statement in lines 3 and 6 can be queried later by a library routine.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Some database languages, such as DMAWK [Sicheram 91], permit a field to have multiple values within a single record. Each field is an implicit zero-based array; the programmer can refer to `FirstName[2]`, for example, to get a person's third name. DMAWK has the strange rule that omitting the subscript represents the last element of the array for R-values, but one past the end for L-values. Assigning a `nil` value to a field deletes the field. Therefore, in a record that is initially empty, Figure 9.40

Figure 9.40

```
FirstName := "Jones"; -- FirstName[0] := "Jones"                    1
FirstName := FirstName; -- FirstName[1] := FirstName[0]             2
FirstName[1] := nil; -- delete FirstName[1]                         3
FirstName := "Hamzah"; -- FirstName[1] := "Hamzah"                  4
```

would have the effect of setting `FirstName[0]` and `FirstName[1]` both to `"Jones"` (lines 1 and 2) before clearing the latter (line 3), later resetting the latter to `"Hamzah"` (line 4).

Since database languages deal heavily with string data, they can take advantage of the data structures and string operations discussed earlier in this chapter, particularly associative arrays and pattern matching. Sal, for example, has both.

3.2 Control Structures

Control structures are needed for setting the current-record mark and for iterating through all relevant records in a relation. Sal has no methods for explicitly moving the current-record mark; it only provides for iteration.

In dBASE, **seek** uses the current order to search quickly for a record whose order value matches the given expression. In addition, the programmer can undertake a search within a subset of the records for one whose fields match any Boolean expression. Such a search is slower than **seek**, because the order information allows an O(log *n*) binary search, where *n* is the number of records. Finally, dBASE provides a **goto** statement that sets the current-record mark to any given record by serial number in the natural order, and a **skip** statement that moves any number of records relative to the current record in the current order. There are predeclared routines that indicate the value of the current-record mark and the number of records in the relation.

Iteration is accomplished in Sal by a **foreach** statement. In Sal, **foreach** indicates which relation variable to use and names a control variable, as in Figure 9.41.

Figure 9.41

```
variable                                                            1
     People : relation                                              2
          FirstName, LastName : string;                             3
          BirthYear : integer;                                      4
     end;                                                           5
     Person : tuple of People;                                      6
```

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

```
put FirstName, LastName                                          7
     into People                                                 8
     from "People.data"                                          9
     where BirthYear < 1990;                                     10
foreach Person in People do                                      11
     if LastName > "Jones" then                                  12
          write(FirstName, LastName);                            13
     end;                                                        14
end;                                                             15
```

Lines 7–10 explicitly copy the external data into an internal variable. It is a runtime error if the declaration in lines 2–5 does not match the contents of file People.data at this time. A **tuple** (line 6) is a record in a relation.

In dBASE, the **scan** statement implicitly uses the currently open relation, as in Figure 9.42.

Figure 9.42

```
open People; -- make the relation available and current          1
filter BirthYear < 1990;                                         2
scan for LastName > "Jones" do                                   3
     write(FirstName, LastName);                                 4
end;                                                             5
```

Natural order is used in lines 3–5, since no **order** statement was encountered.

Nested **scan** statements iterating over the same relation are useful. Figure 9.43 shows how to list all people by age category.

Figure 9.43

```
variable TheYear : integer;                                      1

open People; -- make the relation available and current          2
order BirthOrder; -- increasing BirthYear                        3

scan do -- each iteration covers one birth year                  4
     TheYear := BirthYear;                                       5
     write("During ", TheYear);                                  6
     scan rest while BirthYear = TheYeardo                       7
          write(FirstName, LastName);                            8
     end;                                                        9
     skip -1; -- don't ignore first record of next set           10
end;                                                             11
```

The **rest** keyword on line 7 prevents this **scan** statement from resetting the current-record mark to the start of the relation every time it begins to execute. The **while** clause indicates a stopping condition for this **scan** loop. The surprising code of line 10 is necessary because the **scan** statement of lines 7–9 leaves the current-record mark on the first record that does not match BirthYear = TheYear, but when control returns to line 4, the current-record mark will be advanced again.

Nested **scan** statements iterating over different relations are also quite useful. For example, the code of Figure 9.44 prints the events that occurred in every person's birth year:

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 9.44

```
variable TheYear : integer;                                          1

open People;                                                         2
order BirthOrder; -- increasing BirthYear                            3

open Events;                                                         4
order EventOrder; -- increasing EventYear                            5

use People; -- make relation current                                 6
scan do -- each iteration covers one person                          7
     write(FirstName, LastName);                                     8
     TheYear := BirthYear;                                           9
     use Events; -- ready for nested scan                            10
     seek TheYear;                                                   11
     scan rest while EventYear = TheYear do                          12
          write(What, Place);                                        13
     end;                                                            14
     use People; -- ready for next iteration                         15
end;                                                                 16
```

Because only one relation is current, and the **scan** statements do not remember which relation they are scanning, I need to employ **use** to explicitly reestablish context before each **scan** (lines 6 and 10) and before each iteration (line 15). Luckily, the current-record mark, current order, and filtering information are retained independently for each relation. The **seek** in line 11 moves the current-record mark in Events efficiently to the first relevant record.

It is possible to link the People and Events relations to form a pseudorelation (not persistent) with fields from both, as in Figure 9.45.

Figure 9.45

```
variable ThePerson : string;                                         1

open Events;                                                         2
order EventOrder; -- increasing EventYear                            3
open People; -- natural order                                        4
link Events on BirthYear;                                            5

scan do -- each iteration covers one person                          6
     write(FirstName, LastName);                                     7
     ThePerson := LastName + " " + FirstName;                        8
     scan rest while LastName + " " + FirstName = ThePerson          9
     do -- each iteration covers one event                           10
          write(What, Place);                                        11
     end;                                                            12
     skip -1; -- don't ignore first record of next set               13
end;                                                                 14
```

The **link** statement in line 5 connects the currently open relation, People, with the stated relation, Events, using People.BirthYear (explicitly) and Events.EventYear (implicitly: that is the order field). Each record in the linked relation has fields FirstName, LastName, What, and Place. For every person, there are as many records as there are events that share the same

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

date.

The Sal code for this algorithm, shown in Figure 9.46, has fewer surprises, although Sal has no concept of orders, cannot seek information efficiently, and has no concept of linking relations.

Figure 9.46

```
variable                                                    1
     People : relation                                      2
          FirstName, LastName : string;                     3
          BirthYear : integer;                              4
     end;                                                   5
     Person : tuple of People;                              6

     Events : relation                                      7
          Place, What : string;                             8
          EventYear : integer;                              9
     end;                                                   10
     Event : tuple of Events;                               11

put FirstName, LastName, BirthYear                          12
     into People                                            13
     from "People.data";                                    14
put What, Place, EventYear                                  15
     into Events                                            16
     from "Events.data";                                    17

foreach Person in People do                                 18
     write(FirstName, LastName);                            19
     foreach Event in Events do                             20
          if Event.EventYear = Person.BirthYear then        21
               write(What, Place);                          22
          end; -- if                                        23
     end; -- foreach Event                                  24
end; -- foreach Person                                      25
```

3.3 Modifying Data

Sal is not intended for modifying data (there are related programs for that purpose in the package that contains Sal). dBASE modifies data in the current record by a **replace** statement, which indicates new field-value pairs. Fields that are not mentioned are left alone. New records are added to the end of the relation by an **append** statement, after which it is necessary to **replace** the values of all fields. The current record can be deleted or undeleted; a separate statement is needed to accomplish the fairly expensive operation of physically removing all records that have been deleted and rebuilding order information. dBASE is also capable of copying a relation (or a part of it based on Boolean expressions) to a new relation, with an option to sort the new relation in the process.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

3.4 SQL

SQL (Structured Query Language) was developed during the mid-1970s and introduced commercially in 1979. Since then, it has become widely available. SQL is in a sense a single-minded language: All computation is cast in the mold of relation manipulation. This is just the right level of abstraction for many database operations. I concentrate on expressions that access existing relations; there are also commands that update existing relations. First, Figure 9.47 shows how to compute all people born before 1990 with distinct first and last names.

Figure 9.47

```
select *                                                    1
     from People                                            2
     where BirthYear < 1990 and LastName ≠ FirstName;       3
```

This program fragment is an expression. If it stands by itself, the resulting data are be displayed; it can be placed in an assignment statement or anywhere else that a relation is expected. The * in line 1 indicates that the resulting relation is to contain all fields of the underlying relation, which in line 2 is specified to be People. Line 3 restricts which records are to be selected for the result.

Figure 9.48 shows how to find the names of people whose last name appears after "Jones" and were born before 1990.

Figure 9.48

```
select FirstName, LastName                                  1
     from People                                            2
     where BirthYear < 1990 and LastName > "Jones"          3
```

Figure 9.49 shows how to find all people by age category.

Figure 9.49

```
select FirstName, LastName, BirthYear                       1
     from People                                            2
     orderby BirthYear;                                     3
```

The **orderby** clause in line 3 indicates that the resulting relation is to be sorted by birth year.

The code of Figure 9.50 will print the events that occurred in every person's birth year.

Figure 9.50

```
select FirstName, LastName, What, Place                     1
     from People, Events                                    2
     where EventYear = BirthYear                            3
     orderby LastName + FirstName;                          4
```

This example builds a single relation from multiple relations. Such a computation is known as a **join**. In this case, line 2 specifies that the fields of People and Events are to be combined. Line 3 restricts attention to those records where the EventYear is the same as the BirthYear. Such restriction is common, but not required. It is not necessary to build the restriction out of an equality test. Line 1 restricts attention to four of the resulting fields. Line

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

4 sorts the resulting relation by person. It shows that the sort condition can be any expression. One difference between this code and what I showed previously for dBASE is that people born in years without events are omitted from the result. Another difference is that the result is a relation, which can be manipulated further before printing.

SQL provides several accumulation operators, such as `count`, `sum`, `min`, `max`, and `average`. Figure 9.51 shows how to find the average birth year and the alphabetically last name of all people.

Figure 9.51

```
select average(BirthYear), max(LastName + FirstName)          1
     from People;                                             2
```

Accumulated values are particularly helpful in conjunction with grouping, since the accumulation is computed independently for each group. Figure 9.52 shows how to count how many people were born in each year.

Figure 9.52

```
select BirthYear, Count(*)                                    1
     from People                                              2
     groupby BirthYear;                                       3
```

The * in line 1 refers to entire records instead of a particular field. The result of this expression is a relation with two fields: `BirthYear` and `Count1` (the latter is automatically named). The relation has one record for each distinct value of `BirthYear`.

Individual groups can be suppressed by a **`having`** clause, much as individual records can be suppressed by a **`where`** clause. Figure 9.53 shows how to get the number of people born in each year, but only show those years where the number is greater than 100.

Figure 9.53

```
select BirthYear, Count(*)                                    1
     from People                                              2
     groupby BirthYear                                        3
     having Count(*) > 100;                                   4
```

Expressions can be combined in several ways. The simplest is to take the **`union`** of two expressions. Those expressions must result in structurally equivalent relations (although the names of the fields may differ). Duplicate records are removed. Figure 9.54 shows how to get a relation with all first or last names, along with birth date.

Figure 9.54

```
select BirthYear, FirstName called Name                       1
     from People                                              2
union                                                         3
select BirthYear, LastName                                    4
     from People;                                             5
```

Line 1 introduces a new name for the second field in the result.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

A more complex way to join expressions is by subordinating one to another, as shown in Figure 9.55, which will find those people born after the average birth year.

Figure 9.55

```
select FirstName, LastName                                    1
     from People                                              2
     where BirthYear > average(                               3
          select BirthYear from People                        4
     );                                                       5
```

Line 4 is an expression embedded inside the invocation of average. Similarly, Figure 9.56 shows how to find people born after Ramachandran.

Figure 9.56

```
select FirstName, LastName                                    1
     from People                                              2
     where BirthYear > (                                      3
          select BirthYear                                    4
          from People                                         5
          where LastName = "Ramachandran"                     6
     );                                                       7
```

If there are several records with LastName = "Ramachandran", this expression will fail. In that case, I can modify the expression slightly, as in Figure 9.57.

Figure 9.57

```
select FirstName, LastName                                    1
     from People                                              2
     where BirthYear > any(                                   3
          select BirthYear                                    4
          from People                                         5
          where LastName = "Ramachandran"                     6
     );                                                       7
```

The accumulator any in line 3 allows the **where** clause of line 3 to be satisfied for anyone born after even one of the several Ramachandrans. This accumulator is, in effect, an Icon iterator (described earlier in this chapter). A related iterating accumulator is all; if I had used it in line 3 instead of any, I would only get those people born after all Ramachandrans. Finally, the accumulator exists reduces the result of a subexpression to a Boolean indicating whether the subexpression's value contains any records.

The outer expression can communicate values to the inner expression. Figure 9.58 shows how to find all people born in the year the last occurrence of each event took place.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 9.58

```
select FirstName, LastName                                          1
     from People, Events called OuterEvents                         2
     where BirthYear = (                                            3
          select max(EventYear)                                     4
          from Events called InnerEvents                            5
          where InnerEvents.What = OuterEvents.What                 6
     );                                                             7
```

Lines 2 and 5 give two aliases for Events, so that the two uses of this relation can be distinguished in line 6.

The **select** mechanism of SQL allows programmers to deal with data instead of control structures. The APL language discussed in the next section takes this idea to an extreme.

4 ◆ SYMBOLIC MATHEMATICS

Early languages like FORTRAN were intended primarily for numeric mathematical computation. A completely different class of languages has been developed for symbolic mathematical computation. The major novelty of these languages is that unbound identifiers can be treated as algebraic symbols to be manipulated. The best-known languages in this family are Macsyma, Maple, and Mathematica. These languages can simplify algebraic expressions, perform symbolic integration and differentiation, calculate limits, generate series and sequences, solve systems of equations, and produce graphs. They are almost always used interactively.

Since there are so many different mathematical manipulations possible, mathematical programming languages tend to organize their functions into libraries that are dynamically loaded when they are needed. This organization reduces the amount of memory that a typical session will need. For example, Maple's linear algebra library contains routines for solving linear systems of equations, inverting matrices, and finding eigenvectors and eigenvalues. There are also libraries for combinatorics, for the simplex method, for trigonometric functions, and many other applications. Arrays can be manipulated much as in APL, including extraction of slices in any dimension, so operations like Gaussian elimination are easy to write. In fact, Mathematica has APL's **inner** and **outer** operators.

Figure 9.59 shows some examples of the manipulations possible in these languages.

Figure 9.59

```
in:  poly := 2*x^5 - 3*x^4 + 38*x^3 - 57*x^2 - 300*x+450;          1
     solve(poly=0,x); -- solve with respect to x                   2
out:                    1/2     1/2                               3
     3/2, 5 I, - 5 I, 6   , - 6                                    4

in:  e1 := a + b + c + d = 1;                                      5
     e2 := 2*a + 5*b + c + 4*d = 4;                                6
     e3 := -5*a + 4*b + 5*c - 3*d = -1;                            7
     e4 := b + 4*c - 5*d = 0;                                      8
     SolutSet := solve({e1,e2,e3,e4},{a,b,c,d});                   9
out: SolutSet := {d = 0, c = -2/13, a = 7/13, b = 8/13}            10
```

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

```
in:  f:=x^2 - y^2;                                          11
     diff(f,x);                                             12
out: 2x                                                     13

in:  y^2 + 2*y;                                             14
     factor(%+1); -- % means previous expression            15
out:      2                                                 16
     (1+y)                                                  17
```

The output in lines 3–4 and lines 16–17 is carefully formatted over several lines to look like typeset mathematics. Matrices are also displayed in multiple lines. The identifier I in line 4 is the mathematical constant *i*, the square root of –1. Not only can Maple differentiate polynomials (and other sorts of expressions), it can also differentiate programmer-defined functions, as in Figure 9.60.

Figure 9.60

```
f := procedure (x);                                         1
variable                                                    2
     i : integer;                                           3
     result := 0;                                           4
begin                                                       5
     for i := 1 to 2 do                                     6
          result := result + x ^ i;                         7
     end;                                                   8
     return result;                                         9
end;                                                        10

g := differentiate(f);                                      11
```

Line 11 assigns into g a procedure with the declaration shown in Figure 9.61.

Figure 9.61

```
procedure g(x);                                             1
variable                                                    2
     i : integer;                                           3
     resultx := 0;                                          4
begin                                                       5
     for i := 1 to 2 do                                     6
          resultx := resultx + i * x ^ (i - 1);             7
     end;                                                   8
     return resultx;                                        9
end;                                                        10
```

Much more complicated examples are possible, involving trigonometric functions, for example. The ability to differentiate programmer-defined functions makes it possible to program Newton's method for finding roots of functions, as shown in Figure 9.62.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 9.62

```
findRoot := procedure (f : procedure);                                  1
variable                                                                2
     result := 0; -- or any other initial guess                         3
     epsilon := 0.001; -- or any other desired precision                4
begin                                                                   5
     while abs(f(result)) > epsilon do                                  6
          result := result - f(a)/differentiate(f)(a);                  7
     end;                                                               8
     return result;                                                     9
end;                                                                    10
```

The data types available in mathematical languages include integer, real, arrays, strings, and lists, in addition to symbolic expressions. They also include arbitrarily large integers and fractions of arbitrarily large integers. Maple also provides associative arrays, which are useful for storing values of functions, and arrays with programmer-defined indexing functions, which can introduce structure such as symmetry or triangularity in matrices and can provide default values for arrays at indices that have not been given values. Maple implicitly associates an associative array called the remember table with every procedure. The programmer can request that values computed by the procedure be remembered in that array to short-circuit future evaluations with the same parameters. In other words, dynamic programming is trivial to add to any program, such as the one shown in Figure 9.63 for Fibonacci numbers.

Figure 9.63

```
Fibonacci := procedure[remember](n);                                    1
begin                                                                   2
     Fibonacci(n-1) + Fibonacci(n-2);                                   3
end;                                                                    4

Fibonacci(0) := 0; -- assigns to the remember table                     5
Fibonacci(1) := 1; -- assigns to the remember table                     6
```

The option **remember** in line 1 causes Fibonacci to store and use previously computed values. The assignments in lines 5 and 6 explicitly place values in Fibonacci's remember table, making it unnecessary to put special cases in the body of the procedure itself.

5 ◆ FINAL COMMENTS

Languages that are aimed at special applications tend to concentrate on particular aggregates in order to help the programmer write clear and efficient code. SNOBOL and Icon are particularly designed for applications that need to read and manipulate textual data. The related **scripting languages** are used to scan text files, extract information, print reports, construct input for other programs, and collect output from other programs. Such languages include command interpreters like Csh, stream editors like Awk and Sed, and interpreted languages such as Perl and Tcl. These languages generally have many features for manipulating strings. Extensions to Prolog (see Chapter 8) for dealing with strings are actively being researched, giving rise to languages such as CLP(Σ). The problem that string Prolog must grapple with is

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

that unification over strings is intractable (it is at least NP hard, although it is decidable) [Rajasekar 94]. Which language to use depends, of course, on what is available (Csh is only available under Unix), how fast the program must run (interpreted programs are generally slower), and how sophisticated the string manipulations need to be.

SNOBOL has some excellent points. The fact that backtracking is built into the language frees the SNOBOL programmer from writing backtrack code, which is tricky to get right. Patterns free the programmer from worrying about maintaining an explicit variable for the focus of attention (the position in the subject string that is being matched). Patterns can be assigned to variables and used to build more complex patterns. In fact, the BNF for a context-free (and even a context-sensitive) language can be represented directly in SNOBOL, so it is easy to write parsers.

SNOBOL also has some unfortunate points.

1. There are many ways to build patterns, and it takes a significant amount of effort to learn how to use these methods. Patterns can grow so complex that they become difficult to understand, debug, and maintain.
2. The programmer must remember the difference between pattern-construction time and pattern-matching time. It is easy to write inefficient programs that construct patterns each time they are used instead of saving them in pattern variables. Variables used in a pattern often need to be marked for lazy evaluation.
3. The fact that side effects are an essential part of pattern application makes programs unclear, especially if the pattern is stored in a pattern variable and applied in a different part of the program.
4. Although patterns are something like procedures, they do not take parameters, and they do not introduce a name scope, so they are forced to communicate and perform local computations through global variables.
5. The pattern-matching part of SNOBOL is mostly divorced from the rest of the language. For example, a good way to find if the first comma in a string `Subject` is at least 10 characters from the beginning is shown in Figure 9.64 [Griswold 80].

Figure 9.64

```
Subject match ((break(",") @ here) & fence &                    1
     (delay ge(here,10)));                                      2
```

The '`@`' operator assigns the position in `Subject` achieved by finding the first comma. It is prevented from finding a later comma by the **`fence`** operator. The `ge` integer-comparison procedure is invoked lazily to make sure that `here` is current when the parameters to `ge` are evaluated. This example shows how awkward it is to build programs that involve both pattern matching and arithmetic.

The two novel ideas of Icon, the concept of scanning strings by matching procedures and the idea of iterator expressions, are both unusual and powerful. However, this power has a price. The global nature of `subject` and `pos`, and the fact that matching procedures have side effects on these pseudovariables, can make programs hard to follow. It is possible to directly assign into both `subject` and `pos`, which can wreak havoc, especially in a **`scan`** body. Al-

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

though Icon iterator expressions are as powerful as CLU iterators (and often easier to encode), they are not general-purpose coroutines. They cannot be used, for example, to solve the binary-tree equality puzzle from Chapter 2.

On the positive side, the concept of scanning strings is easily generalized to scanning other data structures, such as trees. A programmer may introduce matching procedures that inspect a subject of any type and modify position variables to indicate progress. Instead of using **scan**, which is specific to `subject` and `pos`, all that is needed is a new name scope with local variables properly initialized, as in Figure 9.65.

Figure 9.65

```
variable                                                    1
     target : ... := ...; -- can be any data structure      2
     position : ... := ...; -- in any representation        3
begin                                                       4
     ... -- expression using matching procedures            5
end;                                                        6
```

In fact, **scan** is just a name scope with variables `subject` and `pos` automatically declared and initialized. It is not necessary to use **scan**, because all predeclared matching procedures have overloaded versions with more parameters that explicitly specify the subject. So everything that is done automatically by **scan** and the matching procedures could be done (maybe with increased clarity) by name scopes, explicit variables, and extra parameters. Some adjustment would be needed to pass parameters like `pos` by reference or value result mode; Icon only has value mode.

Arrays are primarily important in mathematical calculations. However, APL shows that adequately powerful array operations can take the place of control structures; it is possible to build very sophisticated nonmathematical programs in APL. These programs may appear to be inefficient to execute, with very large intermediate results, but clever evaluation techniques allow APL interpreters to work in limited memory. Unfortunately, the programs are difficult to read, especially in the natural APL syntax.

The simplest databases are just ASCII files, with one line per tuple. Scripting languages like Awk, Sed, Perl, and Tcl often suffice to manipulate these databases. More complex databases can be accessed through subroutines in other languages. It is quite common to embed SQL calls, for instance, in a C program. Commercial databases often come with their own languages. dBASE, for example, is a generally Algol-like language interwoven with specific constructs for accessing databases. Paradox, in contrast, is built on an object-oriented model.

Symbolic computation is important to mathematicians and engineers, and especially to students in these disciplines. Languages like Mathematica and Maple allow these users to construct symbolic equations, manipulate them, and view their behavior graphically.

There are other aggregates that I have not covered in this chapter. In particular, specialty languages are very important for statistics, controlling machine tools, and text formatting.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

EXERCISES

Review Exercises

9.1 In Icon, is the expression `tab(4 | 3)` equivalent to `tab(4) | tab(3)`?

9.2 Write a regular expression that matches either `"begin"` or `"end"`.

9.3 Write a regular expression that matches any word starting with `"pre"` and ending with `"ion"`.

9.4 Modify the Icon program of example 21 on page 279 so that the final word in `MyString` may continue to the end of `MyString` without a final space character.

9.5 What is the Icon equivalent of SNOBOL's **fence** pattern?

9.6 In dBASE, it is quite awkward to generate a list of all first names combined with all last names in the `People` relation. Suggest how to manage such a feat.

9.7 Design an SQL expression that builds a relation containing the first name of everyone born before all earthquakes in San Francisco.

9.8 Write a SNOBOL pattern that prints all contiguous substrings of the subject and then fails.

Challenge Exercises

9.9 Referring to Figure 9.5 (page 271), design a variety of `CharSearch` that finds the second `s` in `"sample string"`.

9.10 Write a regular expression that matches all words that can be typed by alternating hands on a standard qwerty keyboard.

9.11 Refer to Figure 9.11 (page 274), and suggest a better component type than `Boolean` for `Present`.

9.12 Write a SNOBOL program that has the same effect as the Icon program in Figure 9.20 (page 279).

9.13 Modify the Icon program of Figure 9.21 (page 279) so that it writes all words that contain telephone numbers, that is, sequences of only digits and an obligatory single hyphen.

9.14 The simple program for `MatchDouble` in Figure 9.23 (page 281) becomes more complex if it doesn't use concatenation. Show how to code it, using neither concatenation nor explicit reference to `pos`.

9.15 Write an Icon program that generates all binary trees on *n* nodes, similar to the ones written in C and CLU in Chapter 2.

9.16 Why is it impossible to write an Icon program that solves the binary-tree equality puzzle of Chapter 2?

9.17 Can Icon iterator expressions and iterator procedures be implemented with a single stack?

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

9.18 Show how to implement `rest` and `last`, mentioned on page 276, using the primitive substring operations.

9.19 Use primitive substring operations to implement Icon's `upto` matching procedure. Don't worry about integrating your result into Icon's backtracking mechanism.

9.20 Use primitive substring operations to build a function `NewBase(x)` that returns a substring that is equal to `x`, has a new base, has left set to the beginning of the new base, and right set to the end of the new base.

9.21 What is the subtle bug in Figure 9.45 (page 295)? How would you fix it?

9.22 Describe what the APL program in Figure 9.66 does.

Figure 9.66

```
in:  n := 30                                          1
     a := 1 ≠ or accumulate (1 1) drop                2
          (transpose 0 =                              3
          (fill n) outer mod fill n) -                4
          (fill n) outer = fill n                     5
     a compress 1 drop fill n                         6
out: 2  3  5  7 11 13 17 19 23 29                     7
```

9.23 In Figure 9.36 (page 289), the **accumulate** operator is represented by an instance of the `Accumulate` class. What would be the response of this instance to a **dimension** query and to a **bounds** query, given that the variable `a` is currently bound to the three-dimensional array with bounds 2 3 4?

9.24 Does it make sense in APL to turn a lazy evaluator into an incremental lazy evaluator?

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.