Contents

FORWARD - Origins of the Clarion Language

by Bruce D. Barrington, CEO, TopSpeed Corporation

As so often happens, I was just trying to please myself. I bought the first PC I ever saw and wanted to program it. That's what I do. Pascal was a straight-jacket and C wasn't available yet. So I tried BASIC. All it needed were some smart screen and keyboard routines. Right? Perhaps a little indexed sequential. Right?

Wrong! I could make it work. But I couldn't make it clean. I had just spent 10 years working with software development tools of my own design. I liked them. Maybe it was time to share what I had learned. Maybe the world really needed yet another computer language—a general-purpose, business programming language. Designed especially for PCs.

It may sound contradictory to call a business language "general-purpose," but in the PC world there are many business "languages" that are anything but general-purpose. Writing spreadsheet macros is programming, I suppose, but the macros hardly comprise a general-purpose language. For that matter, most database languages are not general-purpose languages. They are really scripts to be executed by their database manager. The scripts define a role the database manager plays while acting out your application. Even the dBase language, which can be compiled and run on stand-alone basis, is not really general-purpose.

According to my definition, a general-purpose language should be able to exercise the entire repertoire of capability offered by the underlying platform. That means a program should be able to read any section of any file that is visible to the operating system. It should pass through all the versatility available for the user interface. It should connect, in standard ways, to other general-purpose languages and componentware. A general-purpose language does not contaminate a program with its own "look and feel." It does not erect barriers to be surmounted. Rather, it grants wide latitude within the constraints of its platform to solve a broad range of programming problems with an extensive choice of styles.

But why restrict the new language to PCs? Other mainstream languages are meticulously portable. I decided that PCs deserve special treatment. Even in 1984, when I began designing Clarion in earnest, PCs already comprised a substantial percentage of all the computers installed in the world. And PCs were different than other computers. They were inherently single-user devices with an integrated keyboard and monitor. The keyboard and monitor could be accessed instantly, without modems and communications lines.

These machines begged for responsive, interactive application programs. I wanted to exploit this functionality by building memory-mapped video into my new language. If a Clarion program could “only” run on 40 or 50 million computers, that was all right by me.

I was driven by the steadfast belief that programming should be simpler. That programming languages should be easier to read and write. And that the poor productivity associated with software development stemmed from inadequate and poorly designed programming tools.

These feelings began as pet peeves: Why would anyone design an **IF** statement like **IF**...**THEN BEGIN**;*statements*;**END ELSE**... (Pascal). What possible value do the **THEN**, **BEGIN**, and **END** keywords serve in this structure? Why use “**:=**” instead of “**=**” for an assignment statement (Pascal, Modula-2, Ada). Didn’t the language designer know that assignments are the most numerous statements in a program or that “**:=**” is a finger locking combination of shifted and unshifted keys? How about a **READ**...**AT END** (COBOL) clause that sets an end-of-file variable that is tested to terminate a read loop? Why can’t the loop test for end-of-file? Having declared a variable, why must I remind the compiler to convert it in mixed expressions? Can’t the compiler remember that for me? Have you ever done lint collection? Did you ask why? And hex dumps. What about HEX DUMPS! After twenty years of programming, I felt like the anchorman in the movie *Network* who shouted out the window: “I’m mad as hell and I’m not going to take it anymore.”

Setting the Style

So I set out to design a new computer language that was compact (easy to write) and expressive (easy to read). I began at the back and worked toward the front: First, I wrote lots of programs, experimenting with syntax and semantics until the programs looked great. Then I wrote a small language reference manual. When the manual was well along, the development team started writing a compiler. The language was changing daily. Our old development memos describe an energetic and interactive process. Many ideas were proposed and rejected for reasons of art. Others for poor technology. Some were simply insane. Like Darwin’s species, only the strong survived.

I classify programming languages into three styles: token oriented, sentence oriented, and statement oriented. Token oriented languages like Pascal and C are compact but not particularly expressive. Such languages treat a program as a set of tokens (keywords, data names, constants, punctuation, etc.) separated by “white space” (spaces, CR/LFs, comments, and sometimes commas). The compiler collects the tokens and ignores the white space. Token oriented languages are one-dimensional, so programmers use white space to add a second dimension to their programs:

```
typedef struct {
  unsigned char  Type;   /*the type of structure*/
  unsigned       Vlen;   /*variable length*/
  unsigned char  Dplac;  /*decimal places if decimal*/
  void           *Use;   /*pointer to variable*/
}Usedef
```

This C programmer has done about everything possible to code a readable type definition. But the left brace seems to "dangle" off the **struct** keyword. And Usedef dangles off the right brace. After all, braces aren't very artistic vertical delimiters.

Sentence oriented languages like COBOL and most database languages are expressive but not very compact. Sometimes statements in sentence oriented languages read like perfect English. This COBOL statement is certainly expressive:

```
MULTIPLY PRINCIPAL BY RATE GIVING PAYMENT ROUNDED.
```

But no more so than:

```
Payment = Principal * Rate
```

I would argue that in the context of an entire program, the second statement is easier to read than the first, which tends to melt into paragraphs full of verbiage. Other sentence formats are not very English-like at all. I found this "beauty" in an xBase language reference manual:

```
EDIT  [FIELDS <field list>] [<scope>][FOR <expL1>]
      [WHILE <expL2>][FREEZE <field>]
      [KEY<expr1> [,<expr2>]] [LAST] [LEDIT] [REDIT]
      [LPARTITION] [NOAPPEND] [NOCLEAR] [NODELETE]
      [NOEDIT | NOMODIFY] [NOLINK] [NOMENU] [NOOPTIMIZE]
      [NORMAL][NOWAIT][PARTITION <expN1>][PREFERENCE <expC1>]
      [SAVE][TIMEOUT <expN2>] [TITLE <expC2>]
      [VALID [:F] <expL3> [ERROR <expC3>]] [WHEN <expL4>]
      [WIDTH <expN3>] [[WINDOW <window name1>]
      [IN [WINDOW] <window name2> | IN SCREEN]]
      [COLOR SCHEME <expN4>] | COLOR <color pair list>]
```

Wow! These are certainly English words, but are they expressive? Could any programmer understand an instance of this statement format without a manual? Among many other questions I'd like to ask is: Who designed a **WHILE** clause and a **WHEN** clause in the same statement? It makes me want to scream out the window.

My experimental programs had become statement oriented—that old fashioned style used by FORTRAN and BASIC. Statement oriented languages exploit the fact that source programs are contained in ASCII source files—every line of a program is a record in the file. So record boundaries can be used to eliminate punctuation. I settled on a statement format that proved to be compact, expressive, and versatile:

```
label STATEMENT[(parameters)] [,ATTRIBUTE[(parameters)]] ...
```

Attributes are only used to declare data. Executable statements use the format of a standard procedure call. Of course, I defined different statement formats for assignment statements (A = B) and (IF, CASE, etc.).

A statement label starts in column one (the first position of the record). A statement without a label must not start in column one. A statement is terminated by the end of the line unless it is continued by a vertical bar (|). I adopted the semi-colon as an optional statement separator to allow more than one statement per line. By adopting the Modula-2 concept of ignoring empty statements, I eliminated the distinction between statement separators and terminators that had confounded countless Pascal programmers.

This design eliminates the punctuation otherwise necessary to identify labels and separate statements. Blocks of statements are initiated by a single compound statement such as IF and are terminated by a statement separator such as ELSE (which initiates another statement block) or by an END statement (or period). There are no “dangling” keywords.

Declaring Data

In its infancy, COBOL was said to be “self-documenting” because of its explicit data division and its expressive statement syntax.. Every element that a COBOL program processes must be declared in the data division: variables, constants, files, records, indexes—even sort sequences and report formats. I agreed that these declarations were essential for documenting business programs. And I felt that our new statement format would greatly improve their readability.

In the late 1960’s, IBM promoted PL/I as the successor to COBOL. The language was a disappointment to many, but it did offer a few fresh ideas. By condensing the data type keywords and introducing embedded comments (/*comment*/), PL/I provided enough space to comment every declaration statement. COBOL had been designed for long, descriptive data names. But programmers didn’t use long data names. There were good reasons for this: First of all, programmers like to columnarize programs to make them more readable. Arranging the data division in columns restricts data names to an arbitrary maximum length. Secondly, programmers don’t like long data names in the procedure division. Long names create unwieldy expressions and add to the writer’s cramp produced by an already verbose language. So most COBOL programmers used short, cryptic labels and wrote programs that weren’t nearly as self-documenting as they should have been.

PL/I programmers got around that problem by commenting their declaration statements. If there was a question about the meaning of a data name, it could be resolved by looking up its declaration. I had managed a large PL/I project in the 60’s and became convinced that declaration statements required three parts: a statement label, a data type, and a comment.

The new statement format was perfect. The statement label appeared on the left where it would be most visible. Data type keywords were short (BYTE, REAL, DIM, etc.) to maximize the space available for the comment. As a final space saver, a single exclamation character (!) was designated as a comment initiator.

COBOL and PL/I use “levels” to declare data structures. Every variable has a level number. A variable with a higher level number is “part of” a prior variable with a lower level number. If a variable is not part of a data structure, it is declared as an “01” or “77” level. I never liked using “levels” and was surprised that they were carried over in PL/I. I considered them arbitrary and a waste of space. (What does “77” mean and why do unstructured variables need a level anyway?) I chose GROUP (named after COBOL’s “group item”) as a compound statement to initiate data structures (which we then called “groups”). This mechanism is similar to record...end used in Pascal, Modula-2, and ADA; and struct{..} used in C. Indenting nested GROUP statements produces a very readable declaration:

```
Error   GROUP,PRE(Err)        !Error information
Date      DATE                !Date of error
Time      TIME                !Time of error
Device    STRING(12)          !Active device
Message   GROUP               !Error message
MsgCode     STRING(@P###P)    !Message Code
            STRING(‘ - ‘)
MsgText     STRING(32)        !Message text
          END
        END
```

COBOL and PL/I permit the same data name to be used in different data structures. Such data names are referenced by the data name qualified by the structure name. This is a useful construct, since the same fields frequently appear in more than one data structure (e.g. ACCT-NO IN OLD-VENDOR, ACCT-NO IN CURRENT-PAYEE, etc.). But many programmers refuse to use this feature because it creates such long references. Instead, they code mnemonic prefixes on every field (e.g., VND-ACCT-NO). This takes extra coding time and reduces the available name space.

To deal with this issue, I included an optional prefix attribute that could be attached to any data structure (e.g. **PRE(VND)**). Elements of the structure are qualified by placing the prefix and a colon in front of their data name (e.g. VND:AcctNo, PAY:AcctNo).

To match the functionality of “MOVE CORRESPONDING” in COBOL and “BY NAME” assignments in PL/I, a “deep” assignment statement was added to move matching elements between groups:

```
DestinationGroup :=: SourceGroup
```

As a business language, Clarion needed a rich set of basic data types: All sizes of integers and real numbers were included to provide compatibility with external record layouts and parameter lists. Packed decimals were included to solve rounding problems and reduce memory usage. (They can be declared in a range of sizes.) Various string formats (fixed, Pascal, and C), along with a complete set of string functions, were also included. And finally, data types for dates and times were designed to support direct arithmetic on these variables:

```
Tomorrow = Today + 1
```

But what about structured data types? In ALGOL-like languages such as

Pascal, Modula-2, Ada, and C, groups and arrays are declared as types. You declare the type, then you declare a group or array as an instance of the predeclared type. I never have liked this syntax. In business programs, most groups and arrays are only declared once. Thinking up a type name and coding a **TYPE** statement is usually unnecessary busy-work. I have never considered a group or an array to be a data type anyway. Groups and arrays describe storage relationships, not data types.

So I made the type declaration optional. A Clarion declaration with a **TYPE** attribute declares a data type that can be used for recurring structures or structures that are passed as parameters. A declaration with no **TYPE** attribute declares both a data type and a variable of the same name. I adopted the PL/I **LIKE** statement to declare a variable of predeclared type. I felt that this design offered the best of both worlds:

```
Totals      GROUP,PRE(QTR)
GrossPay      DECIMAL(12,2)
Deductions    DECIMAL(12,2)
NetPay        DECIMAL(12,2)
            END

YTD:Totals  LIKE(Totals),PRE(YTD)
```

Painless Typing

A computer language is strongly typed if every data element has a single data type and the language syntax makes it is impossible to view that element as a different type. Many experts feel that strong typing increases program reliability. Perhaps. But strongly typed programs are harder to write, restricting the use of general purpose procedures, and requiring an unnecessarily vigilant awareness of data types. Furthermore, I have never heard a COBOL programmer accuse **REDEFINES** (used solely to defeat strong typing) of causing reliability problems. (COBOL programmers, by the way, are not uncritical of their language. The **ALTER** statement fell into disuse years ago because it produced unstable programs.)

I didn’t want our new language to be strongly typed. First of all, I wanted to support re-declarations similar to **REDEFINES** or the **union type** in C. Redeclarations are useful for implementing record types (variant records in Pascal) and for handling special programming cases. I assigned the **OVER** attribute to this purpose:

```
MonthNames    STRING(‘JanFebMarAprMayJunJulAugSepOctNovDec’)
Month         STRING(3),DIM(12),OVER(MonthNames)
```

Secondly, I wanted group structures to be treated like strings. This weakened data typing because groups can contain data types other than strings. But groups need functionality. They must be moved, passed as parameters, even (carefully) compared. That’s the rub, of course. Most numeric data types don’t collate as strings, so groups containing numeric elements usually won’t collate properly. Negative integers collate higher than positive integers and floating-point numbers collate somewhat randomly. Design involves compromise (sigh) and I elected the functionality while accepting the risk.

It was important for Clarion data types to permit simple construction of general-purpose procedures. If a procedure expected a numeric parameter, then any numeric data type should suffice. I thought it was ridiculous to require different numeric functions to handle different numeric data types like the ALGOL derivative languages. To go even further, I think polymorphism, as implemented in C++, that requires separate functions for each data type but permits them to be called by a single function name is a notational sham.

In the original version of Clarion, parameters were not even prototyped. Whatever appeared in the callers argument list was used by the procedure. Clarion now requires parameter prototypes but permits the data type to be unspecified. Clarion procedures have always been truly polymorphic for unstructured data.

Clarion parameters are prototyped to be passed by value or by address. Clarion does not support pointers. There are two reasons for this: First, pointers don't carry data type information with them and can be easily misused. And second, pointer dereferences (syntax differentiating the pointer from its target) needlessly complicate programs. It has been my experience that pointer mishaps are involved in most C program bugs.

We chose reference variables, as implemented in C++, to support indirection. A reference variable contains the data type as well as the identity of its target. And a reference variable is automatically dereferenced when it is used. There is no possibility of confusion between a reference variable and its target. Consider the following:

```
CompanyA  FILE
            :
          END
CompanyB  FILE
            :
          END
Company   &FILE              !Company being processed
  CODE
  CASE CompanyLetter         !Which company to process?
  OF 'A'
    Company &= CompanyA      !Point to Company A
  OF 'B'
    Company &= CompanyB      !Point to Company B
  END
  OPEN(Company)              !Open selected company
```

The reference variable *Company* is set by a reference assignment statement (&=). The compiler will object if the data types don't match. Thereafter, a reference variable can be used in any context its target is permitted.

Intermediate Values

Another important issue involved automatic type conversion. I felt strongly that you declared a data type so that the compiler would know! And that an obliging compiler would generate data type conversions as needed. I also felt that a great compiler would probe expressions for meaning and supply logical conversions.

For example, if I add a string to an integer, it is reasonable for the compiler to assume that the string contains an ASCII number and to generate such a conversion. Conversely, if I concatenate an integer to a string, I am asking the compiler to convert the integer first. By selecting appropriate data types for intermediate values, the compiler can safely convert data types in expressions without losing information. If you divide two integers, a good compiler will store the result in an intermediate value that will hold a fraction. If you add an integer to a string, the compiler will also use a fractional intermediate value because a string is capable of expressing a fraction.

Information can be lost, of course, when a value is moved, for instance, by an assignment statement or as a parameter of a procedure call. Moving a real number to an integer truncates the fraction. Moving a real number to a packed decimal rounds to the least significant decimal digit. Some languages, such as Pascal, require that such data conversions be explicitly called. I felt that by declaring a data type, a programmer was requesting the compiler to implicitly restrict the data element to a given domain of value.

Earlier versions of Clarion used just two data types for numeric intermediate values: 32 bit signed integer (LONG) and a 64 bit floating point (REAL). A divide operation or any operation with one or more REAL operands would produce a REAL intermediate value. This strategy provided sufficient accuracy since a REAL could express the maximum numeric significance (15 digits) supported by Clarion. Although they are accurate, floating point values are not discreet. Two equivalent expressions such as 1 / 2 and 2 / 4 can produce floating point results that differ in the least significant bit. This is usually a meaningless difference in computations.

But not in comparisons. A programmer expects one-half to equal two-fourths. I may be willing to avoid comparing REALs but I expect a logical expression such as this to work every time:

```
  IF Hours > Normal * 1.5
```

Using a REAL to receive the expression on the right casts doubt on the results of the comparison. We resolved this issue in Clarion for Windows by implementing fixed-point intermediate values with 31 decimal digits on each side of the decimal point. This change also increased our maximum numeric significance to 31 digits.

Control Structures

While the business languages, COBOL and PL/I, offered the preferred model for declaring data, the ALGOL derivatives, especially Modula-2, offered better control structures. I modified the Modula-2 **IF** statement by making the **THEN** keyword replaceable by a statement separator. This had the effect of eliminating superfluous **THEN**s from multi-line **IF** structures. By adopting Modula-2's **ELSIF,** I eliminated the massive indenting and multiple terminations caused by deeply nested **IF** structures:

```
IF Number < 0
  Sign = -1
ELSIF Number > 0
  Sign = +1
ELSE
  Sign = 0
END
```

I also used Modula-2 as a guide for Clarion's **CASE** statement. Modula-2's **CASE** supports enumerated case labels and case label ranges—very useful features. But I didn't like its punctuation. The **OF** keyword introduces the first case label, but subsequent case labels are initiated by a vertical bar ("|"). I felt this punctuation was ugly and not very intuitive. Instead, I used **OF** to introduce all case labels. I invented the **OROF** keyword to enumerate case labels and the **TO** keyword for case label ranges. These changes produced a very friendly syntax:

```
CASE SUB(Name,1,1)
OF('A') TO ('M') OROF('a') TO ('m')
  DO FirstHalf
OF('N') TO ('Z') OROF('n') TO ('z')
  DO SecondHalf
ELSE
  DO FirstHalf
END
```

Modula-2 was the first usage I had seen of the **LOOP** keyword in its proper context. In Modula-2, **LOOP...END** executes an unconditional loop that is terminated by executing an **EXIT** statement. I augmented this concept by adding a **CYCLE** statement to recycle the loop from within. (I also changed **EXIT** to **BREAK** because I was using **EXIT** for another purpose.) I implemented conditional loops by adding four optional clauses to the **LOOP** statement:

```
LOOP I = 1 TO 100 BY 2
LOOP 10000 TIMES
LOOP WHILE Count > 0
LOOP UNTIL EndOfFile
```

I felt that good program organization required local subroutines. A local subroutine is a block of statements that has been removed from the main logic and is executed by a subroutine call statement. If the subroutine is aptly named, the main logic becomes shorter without losing clarity. COBOL and BASIC use **PERFORM** and **GOSUB** for this purpose. Local procedures in Pascal and Modula-2 nearly fit the bill but they require a prototype statement to declare the parameter types. I didn't want to support subroutine parameters because I wanted all the caller's data to be visible to the subroutine. I designed the **ROUTINE** statement to initiate a local subroutine. **ROUTINE**s are placed at the end of a procedure or function and are executed by a **DO** statement.

A number of languages support executing a single statement from a list of statements as indicated by a statement selection integer. FORTRAN uses the computed **GOTO**. COBOL uses **GOTO...DEPENDING ON**. BASIC uses **ON...GOTO** and **ON...GOSUB**. I wanted to implement a similar capability that would execute any type of statement from a list of statements depending on an integer expression. I named this structure **EXECUTE** after the common **XEQ** machine language instruction which executes the single instruction addressed by its operand. This new structure is, I believe, unique to the Clarion language, but has proven quite useful:

```
EXECUTE UpdateAction
  ADD(Master)
  PUT(Master)
  DELETE(Master)
END
```

Taming the User Interface

In 1970, I was working for McDonnell Douglas Automation Company when we purchased one of the first IV/70 computers built by Four Phase Systems, Inc. It was a marvelous machine—96K of solid-state memory, with a footprint not much larger than a PC. What made this box so interesting was its video support: 32 CRTs daisy-chained from 8 video ports that were refreshed directly from memory. Before the IV/70, every CRT I had used was a communications device. You could watch individual characters display as they arrived at the terminal. With the IV/70, an entire new screen was displayed every thirtieth of a second. It was the perfect platform for interactive programs. But no one seemed to notice. Four Phase was selling the system as a replacement for IBM's clustered CRTs and as a multi-station keypunch.

I had a higher use in mind. In 1973, I formed a company to develop a turn-key hospital information system based on the IV/70 computer. I wrote a multi-user operating system and a macro-language that exercised it. Then I wrote a macro pre-processor and a small hospital information system. The entire process took 9 months.

The macro language accessed the CRTs as if they were memory (that's what they were!) using move macros. The hospital application "painted" the screen by moving literals to the video memory, then placed entry field descriptions in a user field table and returned to the operating system for processing. When a field completed or a special key was pressed, control returned to the application.

This strategy had a distinct operating system “centric” viewpoint. Function keys were connected to screen procedures. Screen procedures created field tables that were connected to field edit procedures. A program didn’t “run” in a conventional sense. In fact, there was no such thing as a program—just a set of procedures that responded to operating system events. The operating system was in control. It was up to the programmer to anticipate its needs. Our programmers eventually became so proficient with this approach that most hospital systems could be designed, implemented, and fully tested before the hardware was cabled together.

But it was never intuitive. Every one of our programmers climbed a steep learning curve. Event-driven programming is hard to grasp. Later, in one of the most vivid flashes of insight I have ever experienced, it dawned on me that an event-driven operating system could be controlled by a conventional program. The user interface would be invoked by a single statement. For Clarion, I called it ACCEPT. The leading edge of ACCEPT would return control to the operating system and the trailing edge could serve as the entry point for all event processing. A small set of functions would be crafted to identify the event that occurred and the fields involved.

Event-driven systems had always seemed “inside-out” to me. I was inside, chained to an oar, obeying the drummer, processing his events. I realized that ACCEPT would make me the master again. Now the drum was mine! I would call the operating system, not the other way around.

But how would Clarion depict a screen layout? Well, if screen literals are data and screen fields are data, then a screen layout has to be a data structure, doesn’t it? I unimaginatively called it a **SCREEN** structure. **OPEN**(MyScreen) would display a screen. **ACCEPT** would enable the keyboard and handles all of the behavior of operator entry. When the operator completes a field or presses a “hot” key, the **ACCEPT** statement would “fall through,” releasing control to the program. **CLOSE**(MyScreen) would restore the state of the monitor before “MyScreen” was opened.

Declaring screen layouts made them easy to process but even easier to design. The development team integrated a screen painter into the Clarion source code editor which could generate **SCREEN** structures. The screen painter could also read **SCREEN** structures. Get the picture? Position the cursor in a **SCREEN** structure and invoke the screen painter. The screen painter interprets the source and displays the screen layout. Now “paint” some changes on the screen and exit. The screen painter changes the source code by replacing the old **SCREEN** structure with the updated version. Interactive visual design like this is impossible without declared structures.

I designed a similar structure for report layouts. **REPORT** structures contain layouts for print lines, page headers and page footers. The **PRINT** statement handles data formatting and page overflow automatically. And a report painter is integrated with the source code editor to maintain **REPORT** structures just like **SCREEN** structures.

Opening Windows

As luck would have it, our user interface design was perfectly suited to Microsoft Windows—an “inside out” operating system if I ever saw one. Windows programmers were having a very difficult time—who could blame them? The “Hello World” example shipped with a popular C++ product was 8 pages long! Windows was in desperate need of a simple messaging model like the Clarion ACCEPT loop. We decided to provide just that.

We changed our SCREEN structures to WINDOW structures, introducing the grammar necessary to declare and contain Windows objects and properties. We added multi-threading to accommodate the multiple document interface. We changed the grammar of REPORT structures to depict WYSIWYG reports, background forms, and nested group headers, footers, and sub-totals.

The ACCEPT statement became a structure defining the boundaries of an event processing loop. We designed the compiler to cooperate with the run-time library to hide the direction of the procedure calls used to process window events. A call to a run-time window processor is generated above the ACCEPT loop. The loop itself is generated as an embedded accept procedure.

The window processor creates the necessary objects, specifying a common event processing procedure for every event produced by every object. This event processor handles “housekeeping” events such as redraws and calls the embedded accept procedure to deal with other events. When the window closes, the window processor returns control to the statement following the ACCEPT loop.

To the Clarion programmer, it is all quite simple. Open a window, then fall into an ACCEPT loop. The ACCEPT loop cycles for every event the program needs to see. Close the window and fall out of the loop.

We defined a convenient set of functions to identify the events and objects involved. The code necessary to process a typical dialog box looks like this:

```
OPEN(Window)                 !Open the window
ACCEPT                       !Enable the window
  CASE FIELD()               !Which field needs attention?
  OF ?OK                     ! 'OK' needs attention
    CASE EVENT()             !  Which event has occurred?
    OF EVENT:Selected        !   'OK' is pressed down
        :                    !    Process the OK button
      CLOSE(Window)          !    Close the window
    END                      !  End CASE EVENT()
  OF ?Cancel                 ! 'Cancel' needs attention
    CASE EVENT()             !  Which event has occurred?
    OF EVENT:Selected        !   'Cancel' is pressed down
        :                    !    Process 'Cancel' button
      CLOSE(Window)          !    Close the window
    END                      !  End CASE EVENT()
  ELSE                       ! Must be a non-field event
    CASE EVENT()             !  Which event has occurred?
    OF EVENT:CloseWindow     !   The window will be closed
        :                    !    Process window close down
    END                      !  End CASE EVENT()
  END                        ! End CASE FIELD()
END                          !End ACCEPT
RETURN                       !Return to the caller
```

A by-product of our object-oriented run-time library corrected a serious deficiency in the Clarion language—compiler invariants. Declaring screens, reports, and files is very illuminating. But it can also be restrictive. Because they are compiled in, you can't change most declarations at run-time. Many of the language extensions requested by Clarion programmers involved making declared attributes visible to and changeable by the program

In our Windows run-time library, these structures are objects. Objects have properties. And properties can be changed. Anytime. Since we had already overloaded the period as both a structure terminator and a decimal point, we could not implement the standard object oriented notation of *object.property*. So we elected to use "curly brackets" to enclose properties. With this notation, any declared attribute, such as the text displayed on a button, can be modified by a statement such as:

```
?Button{PROP:Text} = 'My Button'
```

Designing a Database

I wanted to implement a simple database syntax that would support all three standard file access methods, direct, sequential, and indexed. The underlying file organization would also be simple: The file would contain a header followed by fixed length data records. The header would describe the record layout and associated keys and memos which would reside in separate files. This arrangement is similar to that used by dBase—a record could be accessed sequentially or directly by key or by its relative record number. I designed a **FILE** structure, similar to a COBOL **FD,** to declare files and their components:

```
Detail    FILE,PRE(DTL),NAME('C:\LEDGER\DETAIL.DAT')
AcctKey     KEY(DTL:AcctNo,DTL:Period,DTL:Date)
BatchKey    KEY(DTL:Batch,DTL:Period),DUP
Comment     MEMO(4096)
            RECORD              !Detail record
AcctNo        SHORT             !Account number
Period        BYTE              !Accounting period
Date          DATE              !Transaction date
Batch         STRING(12)        !Batch ID
Amount        DECIMAL(12,2)     !Amount (+/- = debit/credit)
            END
          END
```

I implemented sequential processing using **SET**, **NEXT**, **PREVIOUS**, and **SKIP** verbs. **SET** establishes the sequence (by key or relative record number) and starting point for the other three verbs which read records forward and backward, and skip over records. These verbs combine nicely with the end-of-file function (**EOF**) in a read loop:

```
SET(Dtl:AcctKey)                !Set account number sequence
LOOP UNTIL EOF(Detail)          !Loop through every record
  NEXT(Detail)                  !Read the next record
    :
END
```

The **GET** verb reads a record randomly by key or relative record number. Importantly, **GET** does not interfere with sequential processing by resetting the next record processed. **PUT** and **DELETE** process records accessed by **NEXT**, **PREVIOUS**, or **GET**. **ADD** inserts a new record in the database. This database access grammar proved to be efficient, robust and versatile—an essential and popular component of our product.

As the Clarion language spread, however, it took on new responsibilities. Clarion developers needed to access dBase files. So we added a dBase procedure library (we called Clarion procedure libraries "Language Extension Modules"—or LEMs). Then Novell came out with client-server support for Btrieve (server-based indexing). Some large Clarion applications needed Btrieve to improve their transaction throughput. So two of our third-party developers came out with Btrieve LEMs.

That left DB2. And RDB. And Oracle. And SQL Server. And every other variety of database that runs on or is accessed by PCs. We were planning to support direct C function calls in the next version of the language, so any database with a C language API could be accessed by a Clarion program. But it was clear to me that this was not the answer. Surely a general-purpose business language shouldn't be using a different grammar for every database format. Migrating a data file shouldn't require a major program overhaul. The Clarion language needed standardized, built-in support for all common databases.

It was suggested that we adopt SQL as our database grammar. I took the suggestion seriously and rewrote some typical Clarion programs using embedded SQL. It wasn't long before I realized this was a terrible idea. When used as a programming language, SQL is extremely verbose and inelegant. The little four statement record loop illustrated earlier becomes this albatross under SQL:

```
DECLARE X CURSOR
  FOR SELECT     *
    FROM         Detail
    ORDER         BY Dtl:AcctNo,Dtl:Period,Dtl:Date
  END
END
OPEN X
LOOP
  FETCH X
  IF ReturnCode = 100 THEN BREAK.
    :
END
CLOSE X
```

Not only are SQL cursors inelegant, they are also nearly useless. You can't make a cursor skip—for example, to re-display a prior page of records. And you can't make it relocate—for example, to jump to "Jones" while browsing alphabetically. I concluded that if I were to replace the Clarion database access syntax with SQL, I would have been tarred and feathered and run out of town on a rail.

So we decided to implement replaceable database drivers. Clarion programmers liked their database grammar, they just needed support for other database formats. By building on the existing language structure, we would be leveraging their knowledge as well as enhancing their current applications. With our new database driver technology, we would make all databases look alike—a non-trivial benefit.

A New View

To produce SQL database drivers, we map SQL syntax onto our own database grammar. Our **SET** statement constructs an SQL **SELECT** statement which is issued at the first instance of a **NEXT** or **PREVIOUS** operation. If you change directions (e.g. **NEXT**...**PREVIOUS**), the driver issues another **SELECT** with a different **ORDER BY** clause. Our **GET** issues a **SELECT**...**FETCH**. **ADD** issues an **INSERT**; **GET**...**DELETE** issues a **DELETE**; and **GET**...**PUT** issues an **UPDATE**. A few features, such as relative record access, are not supported for SQL databases, but otherwise, the implementation is quite complete.

However, our database grammar was unable to exercise some very important SQL features. Clarion programs implement record filters by reading and throwing out unwanted records:

```
LOOP UNTIL EOF(Part)
  NEXT(Part)
  IF Prt:OnHand > 0 THEN CYCLE
    :
END
```

An SQL database can filter records on the server and save a lot of time. Clarion programs join files by reading the primary record to prime a key in order to read the secondary record. An SQL database returns the primary and secondary records with a single access. And Clarion programs read every field in every record on every access. SQL returns only the fields you need.

Of course, an SQL database cannot read minds. You have to tell it what you want it to do. So we designed a VIEW structure for this purpose:

```
View  VIEW(Part),FILTER('PRT:OnHand = 0')
        PROJECT(PRT:Number,PRT:Name,PRT:OnHand,PRT:Usage)
        JOIN(Vendor,PRT:Vendor,VND:Number)
          PROJECT(VND:Name,VND:Address,VND:CityStateZip)
        END
      END
```

This VIEW structure consolidates the intentions of a Clarion program so that the database driver can utilize any services offered by its underlying database engine. The database driver either performs filter (record selection), join (record lookup), and project (field selection) operations or requests the database server to do so. In either case, performance is optimized.

There was also a problem implementing optimistic concurrency under SQL. To update a shared file, a Clarion program reads and saves a record. Then, before it is updated, the record is locked, reread, and compared to the saved copy. If they are the same, the changes are written to the database. Otherwise, the record has been changed by another workstation and the operator is so advised. This process is called “optimistic concurrency” and is based on the expectation that records are usually unchanged.

SQL implements optimistic concurrency with a WHERE clause that requires that all fields to be updated continue to have the same value. If one or more fields have changed, SQL returns an appropriate error. Since Clarion had no syntax to make such a request, we added a WATCH statement for this purpose. WATCH is issued before a GET, NEXT, or PREVIOUS to initiate optimistic concurrency. When the record is accessed, the driver saves a copy. In response to the PUT statement, the driver either rereads the record for comparison or issues an UPDATE...WHERE. to an SQL database. If the record has changed, PUT returns an error.

Our First Compiler

We shipped version 1.0 of Clarion in May of 1986 with both a compiler and an interpreter. The Clarion Compiler produced intermediate code that was then interpreted by the Clarion Processor. The intermediate code was so compact, that large Clarion applications would run on the small memory sizes (256K) that characterized PCs of that era. The compiler produced such tight code by generating a binary description of every declaration statement. Then the data was addressed by a two-byte pointer to the binary description. So it took five bytes to add an integer to a string and format the result according to a picture (one byte for the add operation and four bytes for the pointers to the integer and picture string descriptions). For every operation, the Processor examined the data types of the elements involved and performed any necessary conversions.

But tight intermediate code wasn’t the primary reason for this design. By interpreting the output from the compiler, the Processor could execute a Clarion application without requiring a link step. This was no small consideration. In 1985 and for a long time thereafter, linking was a time-consuming process. Our customers appreciated quick testing, but they also let us know that “real” programming languages produced **.**EXE files! Early the next year, we released the Clarion Translator that converted Clarion intermediate code into **.**OBJ files by replacing the operation codes with procedure calls. The pointers were passed as parameters. This strategy served us well for six years but also posed some problems:

- We had trouble with external libraries. **.**OBJ files could be linked into a Clarion **.**EXE, but they could not be executed directly by the Processor. We designed a process that converted a suitable **.**OBJ into a special binary format (LEM) that could be executed by the processor and changed back into an **.**OBJ by the Translator. But the process was complicated and was only used by sophisticated developers.
- Simple Clarion programs produced big **.**EXEs. The run-time decision making referenced library procedures that were included in the **.**EXE but never called. That made a “Hello World” program take 141K.
- Clarion applications ran slower than C, Pascal, and Modula-2 programs because Clarion programs examined data types at run-time while the other languages did so at compile time.
- It was no longer necessary to avoid linking in the test cycle. New linkers that supported run-time libraries could link a program for testing as fast as we could load the Processor.

Most importantly, we needed technology that would provide a development path to Windows, protected mode, OS/2, UNIX, 32-bit, and non-Intel architectures.

A New Partner

In May of 1990, we solved those problems and many others by licensing the TopSpeed technology from Jensen & Partners International (JPI), a British company. JPI was formed in 1988 when Niels Jensen, founder of Borland International, and his language development team left that company as a group. They purchased their work in progress and produced the TopSpeed product line, the top-rated compiler technology in the industry. JPI had developed C, C++, Pascal, and Modula-2 compilers that shared the same optimizing code generator and project system. JPI called the compilers “front-ends” and the code generator the “back-end.”

We started immediately writing a Clarion front-end. As usual, it was harder than we thought. The language required more changes than we expected. The project took longer and used more resources than we thought it would. But we were thrilled with the results.

We knew the TopSpeed back-end was good, but we were astonished when a Clarion “Sieve of Eratosthenes” (an algorithm for finding prime numbers) ran twice as fast as the same program written with Borland’s Turbo C++. We had also licensed TopSpeed linking technology, but I hadn’t realized just how good it was. TopSpeed’s unique ”Smart Linking” produced perfect granularity by eliminating all unreferenced procedures and static data elements from an **.**EXE. Better yet, while we were working on our front-end, JPI had developed an automatic overlay loader, DOS DLLs, a royalty-free DOS extender, and had announced 32-bit support. With this state-of-the-art technology, we had finally removed the performance penalty that had always been associated with high-level business languages.

In September of 1991, we announced our new product at the first Clarion Developers Conference. New features and the Clarion/TopSpeed connection drew rave reviews. Caught up in the festivity of the occasion, Niels Jensen, and I started talking about merging our companies. It made a great deal of sense. TopSpeed products would gain a US presence and access to a much larger programming market. Clarion products would own their core technology. We would be the first to apply leading edge compiler technology to business software development tools. After a lengthy negotiation, the merger was concluded in April of 1992. Two and a half years later, after the companies had completely homogenized their operations and product lines, the successor company was renamed TopSpeed Corporation. In October of 1994, TopSpeed Corporation released Clarion for Windows, the first product developed in its entirety by the merged companies.

Where We Stand Now

These remarks originally comprised the introduction to the *Programmer’s Guide* that accompanied Clarion Database Developer Version 3.0, released in April of 1993. Extensive additions and revisions have been necessary for this version. Such is progress. I think of software development as the process of gently rocking a Chinese checker board until all the marbles fall into holes. I believe in the notion of a final, correct design. Until Clarion for Windows, I felt that we were a long way from our goal. Now I am not so sure. There are very few marbles rolling.