Ada+SQL – An Overview

Arthur Vargas Lopes
Arthur Vargas Lopes & Co. LTDA
Universidade Luterana do Brasil
Rua Inhanduí 303, 204
90.820-170 Porto Alegre, RS - Brazil
avl@zaz.com.br

ABSTRACT

Ada+SQL is a programming environment for Ada 95 extended with basic SQL single user capabilities. It incorporates a very fast compiler and interpreter, with debugging options, library generator and browser, syntax template editors, programmer wizard, SQL interactive interface and hypertext documentation on the environment, Ada 95 and SQL. Several implementation aspects are discussed.

KEYWORDS

Compilers, interpreters, programming environments, relational databases, computer education.

1. Introduction

Ada+SQL is programming environment (PE) aimed to novice Ada and SQL programmers that have been in development since November of 1993. The PE is an intuitive environment from which the user has easy access to an editor, compiler, interpreter and other tools. Ada capabilities available cover about 90% of the core language specification. In addition, a set of packages have been added to easy programming tasks such as screen and 2D graphics operations. A MS-DOS console controlled by the PE, which runs on Windows 95 and NT, carries out compilation and execution of programs, including SQL commands. The compiler and interpreter are assembled into one independent 32 bits program. Therefore, it is also possible to compile and run programs from the MS-DOS command prompt. Previous versions were distributed over many Internet hosts across dozens of countries and hundreds of several types of institutions accounting many thousands of ftp transfers from whom valuable feedback was received. These versions were known as AVLAda9X and later AVLAda95. Ada+SQL is programmed in ANSI C and visual C++ and contains over 190,000 lines of code. The executables run under both Windows 95 and NT platforms with at least four MB of available RAM. Five MB of disk space is needed to store the Ada+SQL distribution files.

2. Programming Environment

The PE is a windows multiple document interface application named E95. The *SQL* main menu option will be shortly described below. Figure 1 shows one activation of E95.

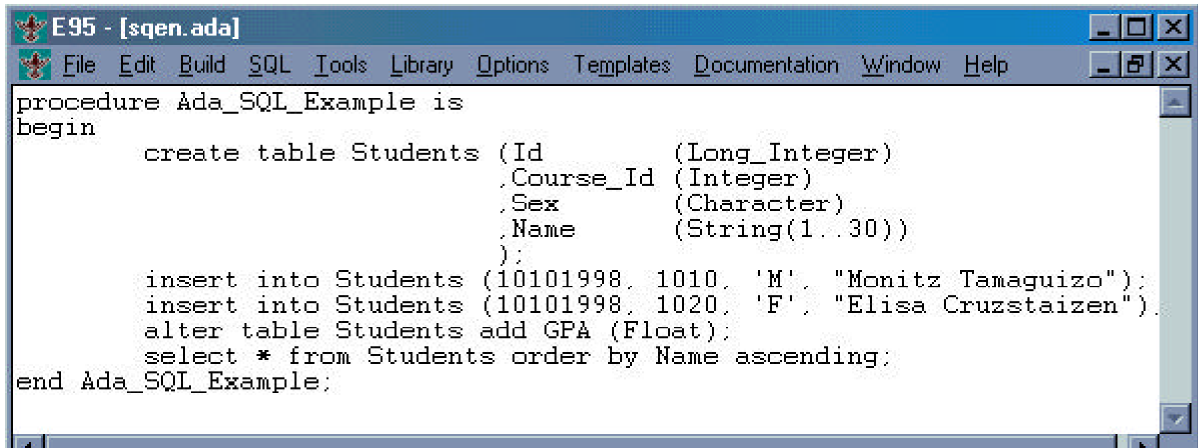

Figure 1 – The Ada+SQL Programming Environment

One editing window shows an Ada program that uses SQL commands. As can be seen the Ada 95 programming language has been extended with SQL commands and the SQL create command has been extended with Ada 95 types. The program output for figure 1 is shown in figure 2.

```
ID        COURSE_ID SEX NAME                             GPA
10101998  1020      F   Elisa Cruzstaizen               0.00
10101998  1010      M   Monitz Tamaguizo                0.00
```

Figure 2 – Output of Program Ada_SQL_Example

This allows the user to either write Ada code intermixed with SQL commands, as shown in figure 1, or to execute a set of SQL commands interactively from the PE. A specialized dialog box, activated from the SQL main menu option, allows the specification and execution of SQL commands. This dialog box is shown in figure 3.

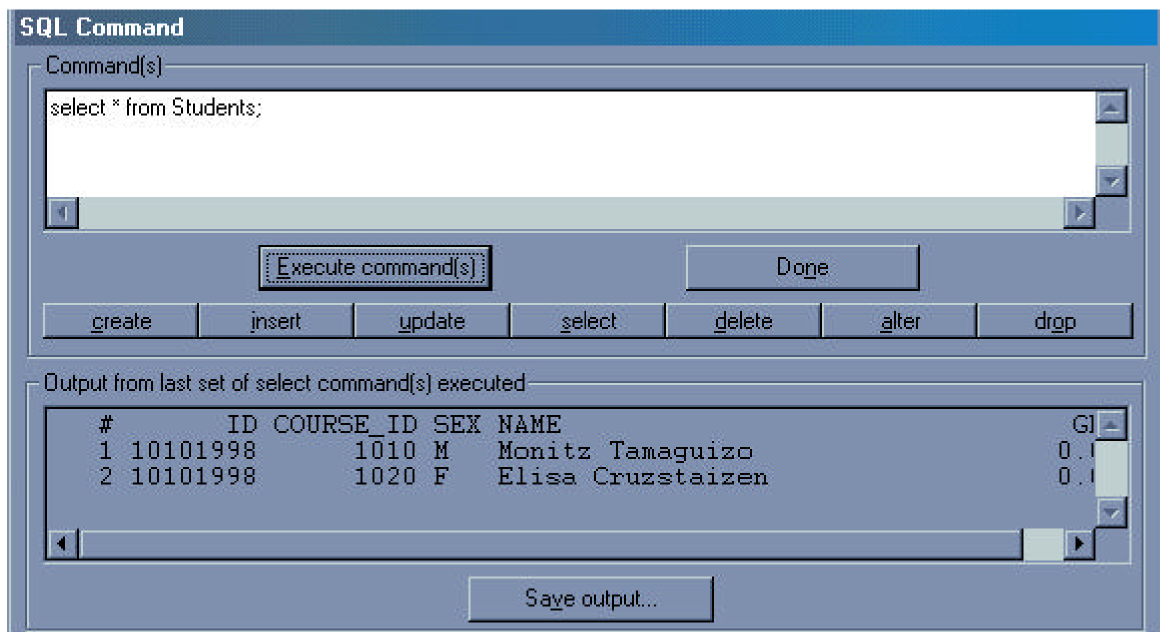

Figure 3 – SQL Command Dialog Box

To assist on the specification of basic SQL commands a set of push buttons (*create*, *insert*, *update*, *select*, *delete*, *alter* and *drop*) are available. Each push button activates a specific SQL syntax template dialog box. Figure 4 shows the SQL Select Command Syntax Template dialog box. The *Append to command area* push button may be used to append the formatted dialog contents to the SQL Command dialog box Command(s) editing area.

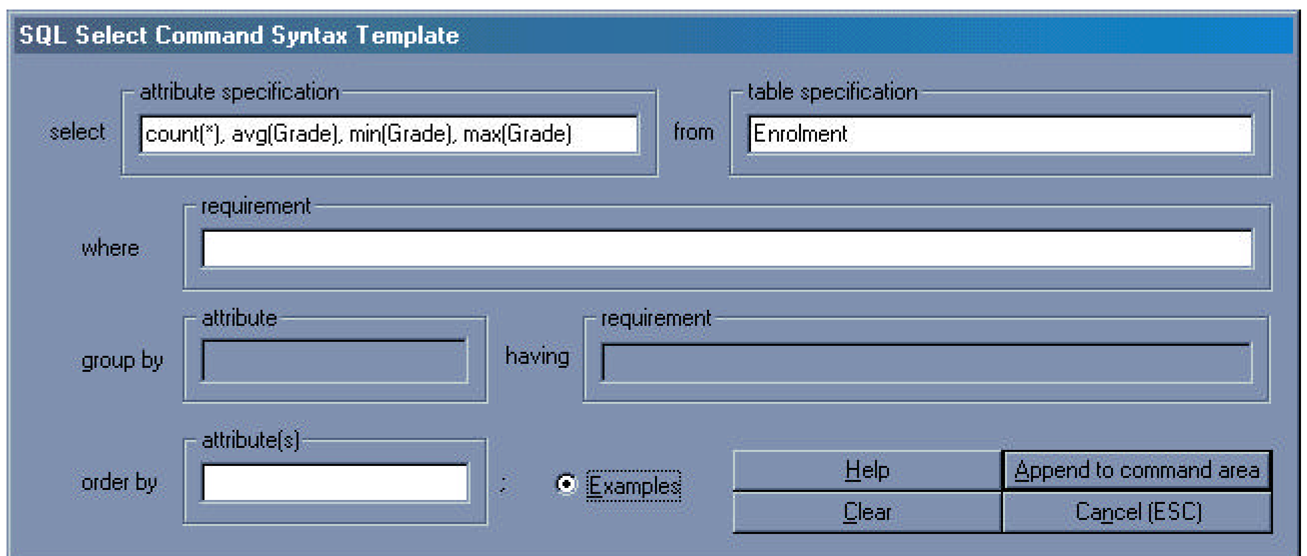

Figure 4 – SQL Select Command Syntax Template Dialog Box

These syntax templates are available to help novice programmers to code basic SQL commands. By pressing the *Help* push button the user is given direct access to related SQL information.

3. Compiler Structure

Compiler components include the usual basic modules such that scanner, parser, symbol table management, error handling, and intermediate code generation. Besides these components a library module keeps track of source files and the program units they contain. The library manager tells the compiler the source file that contains a given program unit such that a package. A source file may contain any number of program units. For instance, when a compilation *A* finds a *with clause* that refers to a package *P,* the compilation makes a pause with the current source file, *a*. A new compilation, *B*, is started to process source file, *b*, which contains package *P*. When compilation *B* concludes, then compilation *A* resumes now accumulating the program units stored in source files *b* and partially *a*. Therefore, no intermediate code is ever stored. Only source code is used.

3.1 Scanner

Lexical analysis operates mostly under parser’s control. Some subprograms belonging to predefined packages also use some scanner operations at run-time. To increase compiler speed, the entire source file is read into a memory buffer by one low-level input system call. A token look-ahead buffer of varying length is used to provide the parser with a means of inspecting the source code in order to identify specific language constructs. Examples of such constructs are aggregates and SQL commands. The use of context clauses and generic units cause the parser to switch its focus among a number of source files.Subprograms that store all scanner information into its local storage are used to deal with this problem. The scanner parameters are then initialized with the information needed to compile the new coming compilation and the parser is then called to act on it. When the parser returns these subprograms restore the scanner information with the previously saved local storage allowing the parser to resume its work with the source file that caused this type of action.

3.2 Parser

Syntax analysis is performed by a recursive descendant parser (*k*) where *k* stands for any number of look-ahead tokens. As pointed out before, language constructs such as aggregates and SQL commands require a number of look-ahead tokens in order to be identified properly. SQL commands are parsed by context. SQL commands may be also coded anywhere an Ada statement is allowed. As mentioned before, instantiation of generic units cause the parser to save the scanner state and redirect its input to the generic unit components making text replacements among generic formal parameters and generic actual parameters. Internal code generated is associated with information on both source file and line number that originated each instruction (see figure 8).

When either a compile or run-time error is detected this information is used to report the exact error location. The SQL select command was extended to allow, as an option, the inclusion of block. Each time a select command retrieves data, that satisfy a where clause when present, the block is executed.

3.3 Symbol Table

The symbol table uses a nested hash table scheme. Each compiler instance contains a global hash table. Program units such that packages, main programs and SQL tables are common entries within this scope. These types of entries have descendant hash tables. For instance, a descendant hash table for a procedure will contain all elements that belong to its declarative part. Figure 6 shows the symbol table representation considering the compilation for the code fragment within figure 5. The outer square represents a scope. Each scope contains 20 fields. Only two of these fields are being shown: the *father* and the *bucket* vector. The *father* field stores a pointer to the ancestor scope. Each *bucket* entry points to a linked list of table entries that share the same hash value. Figure 6-(a) shows the entries *Ada* and *ST*. A full symbol table entry contains 65 fields from which many are pointers into other data structures. Only two of these fields are being shown: the *id* and the *son*. The *id* field stores the characters that represent the symbol. Names, all forms of literals and temporaries are common symbol table entries. The *son* field stores a pointer into its descendant scope for entries like packages subprograms and records. Figure 6-(b) shows the descendant scope for procedure *ST*. Figure 6-(c) shows the descendant scope for package *Ada*. Experiments have shown that this structure is responsible for part of the compiler speed.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure ST is
      type Date is record
            Month : Integer;
            Day   : Integer;
            Year  : Integer;
            end record;
      function Fat(N : Integer) return Long_Integer is
      begin
```

Figure 5 – Fragment of an Ada 95 Program

Pre-defined package specifications are only stored within the symbol table when refereed.

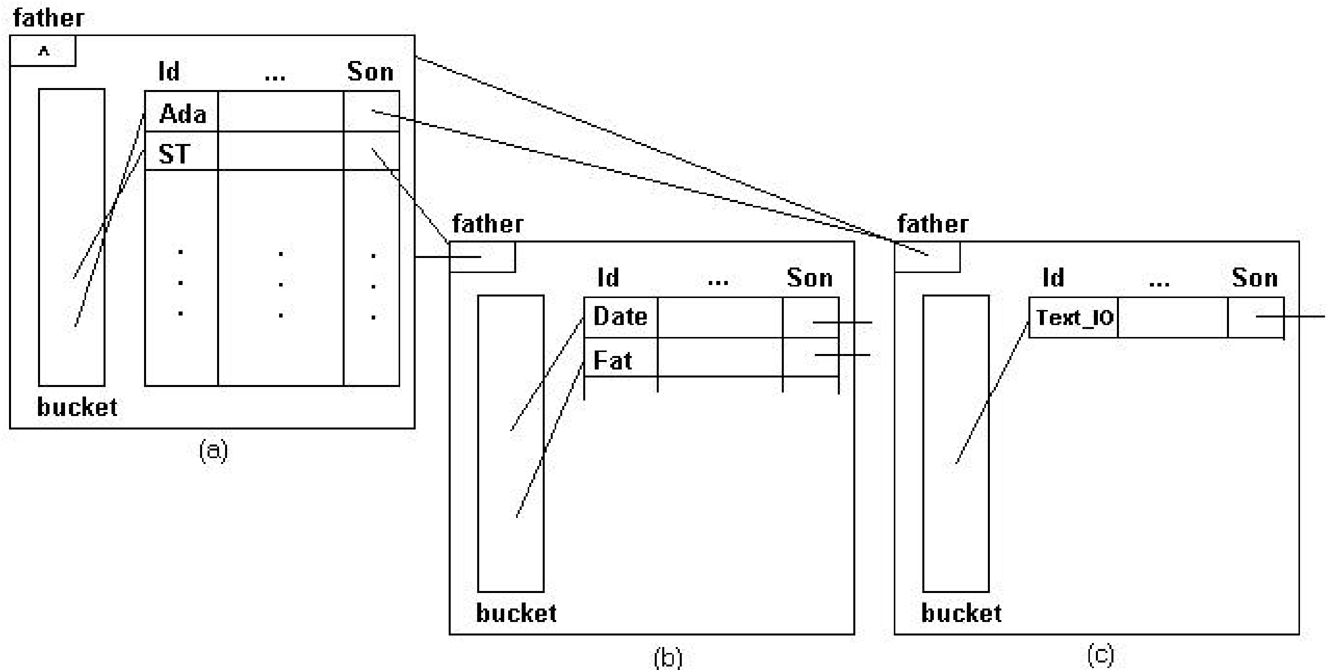

Figure 6 – Partial Symbol Table Representation

3.4 Error Handling

A simple strategy is used for dealing with errors. Once an error is detected it is reported and the compiler terminates. Over one thousand messages are used in both English and Portuguese. A number of external programs are executed to exchange information with the compiler and the PE. If the compiler is executed from the MS-DOS prompt the error message is displayed and the compiler terminates its execution. Otherwise, the compiler is executed from within the PE. In this case, a windows based program displays the error message and additional information. Whenever possible, the PE is positioned with the focus on the editing text window that contains the source line that caused the error.

3.5 Intermediate Code

The intermediate code is based on a set of 163 instructions. Figure 7 shows the layout for the five types of instructions. Calls for built-in subprograms use a set of one-operand instructions. Each actual parameter generates a *push address* instruction. An additional *push address* instruction is used for a function's return value. The actual subprogram call use a one-operand instruction. The operand field, for this instruction, contains the memory address of the built-in subprogram. *Unconditional branches*, *enter block*, *leave block*, *return* and other specialized instructions use only one operand. *Assignment* statements, conversions, *goto if true* instructions use two operands. Examples of three operand instructions are related to the *first*, *last*, *length* and *range* attributes.

8 Bits	16 Bits	
Op Code	Source File	
	File#	Line#

(a) Without Operands

8 Bits	16 Bits		32 Bits
Op Code	Source File		Operand 1
	File#	Line#	

(b) One Operand

8 Bits	16 Bits		32 Bits	32 Bits
Op Code	Source File		Operand 1	Operand 2
	File#	Line#		

(c) Two Operands

8 Bits	16 Bits		32 Bits	32 Bits	32 Bits
Op Code	Source File		Operand 1	Operand 2	Operand 3
	File#	Line#			

(d) Three Operands

8 Bits	16 Bits		32 Bits	32 Bits	32 Bits	32 Bits
Op Code	Source File		Operand 1	Operand 2	Operand 3	Operand 4
	File#	Line#				

(e) Four Operands

Figure 7 – Lay-out of Intermediate Instructions

Four operand instructions are used to implement SQL constructs such as the one shown in figure 8, part (b). Instruction number 48 shows an intermediate instruction that implements the where clause shown in part (a).

```
select * from Grades where Obs is not in ("AP", "IN");
```

(a) Select Command

Instruction#
Source file#
Source line#

```
44  0  7 I_SQL_OPEN_WRITE     $Select_1 7241b8
45  0  7 I_SQL_SELECT_START   NOTAS $Index_GRADES_1
46  0  7 I_SQL_EOF            $Index_GRADES_1 goto 60
47  0  7 I_SQL_READ           NOTAS $Index_GRADES_1
48  0  7 I_SQL_IS_IN_SET      OBS $SQL_SET_IS_IN_1 $SQL_SET_is_in_1 1
49  0  7 I_SQL_NOT            $SQL_SET_is_in_1 goto 58
50  0  7 I_SQL_ADD_ONE        $Select_1
     ...
57  0  7 I_SQL_WRITE          $Select_1
58  0  7 I_SQL_NEXT           GRADES $Index_GRADES_1
59  0  7 I_DIRECTGOTO         46
```

(b) Intermediate Code

Figure 8 – Intermediate Instructions

4. Interpreter

The symbol table and the intermediate code represent any set of compiled program units. This representation together with a non empty set of run-time stacks are used by the Ada+SQL abstract machine which is implemented by an interpreter. Each task and

protected units have their own stack. The tasking model implementation uses a round robing scheduler. The user can change the quantum as a way of experimenting different concurrent program behavior. When a dead state is detected among a set of tasks these tasks are placed in the abnormal state and the exception Tasking_Error is raised. The stack model is used as the run time storage management strategy. Two exceptions are the allocation of dynamic memory, requested explicitly by the programmer, and the occurrence of unconstrained arrays. For instance, the bounds of an unconstrained array declared within a subprogram may be taken from parameters. Only when the subprogram is executed the array storage area is allocated. These arrays make use of special descriptors that point to memory allocated from the heap. Special memory controls are used to avoid memory leaks when a block containing such arrays leaves the stack. SQL attributes are members of a hash table that is a son of its table entry. Each attribute is associated with a position within the master task stack. A stack position of an attribute or Ada object is determined from its level and offset plus the current level of the program unit that embodies its reference.

5. Conclusions and Future Work

Ada+SQL have been used as a tool to assist novice programmers enrolled in Data Structures and Concurrency classes for several years at undergraduate CS programs in the “Universidade Luterana do Brasil” located in the cities of Canoas and Gravataí both within the state of Rio Grande do Sul, Brazil. Students’ reactions indicate that Ada+SQL has been helpful to them. Future work will continue to implement more Ada 95 and SQL constructs as well as improving the PE. Special attention is being given to incorporate some Visual Basic features that make its usage easier. Version 3.0 of Ada+SQL will be distributed upon request over attached e-mail for a limited period of time.

Selected References

[1] Hawryszkiewycz, I. T. Database Analysis and Design. Science Research Associates, 1984.

[2] Winderhold, Gio, Database Design. McGraw-Hill, 1977.

[3] Bradley, J. File. Data Base Techniques. CBS College Publishing, 1982.

[4] Date, C. J. An Introduction to Database Systems. Addison Wesley, 1977.

[5] Date, C. J. and Darwen, Hugh. A Guide To Sql Standard. Addison Wesley, 1996.

[6] Ladanyi, H. Sql Unleashed; The Comprehensive Solutions Package With Cdrom. Sams, 1997.

[7] Freeze, Wayne. The Sql Programmer's Reference, With Cd. Ventana Press, 1998.

[8] Ada 95 Reference Manual. Intermetrics, Inc., Cambridge, Ma, 1995. ANSI/ISO-8652:1995.

[9] Ada 95 Rationale - The Language - The Standard Libraries. Intermetrics, Inc., Cambridge, Ma, 1995.

[10] Bacon, J. Concurrent Systems, Addison-Wesley, 1996.

[11] Ben-Ari, M. Principles od Concurrent Programming, Prentice-Hall, 1982.
[12] Cormen, T. H. et al. Introduction to Algorithms. 15 ed. McGraw-Hill, 1995.
[13] Aho, A.V. et al. Data Structures and Algorithms, 28 ed. Addison-Wesley, 1987.
[14] Aho, A.V., Sethi, R. and Ullman, J. D. Compilers, Principles, Techniques, and Tools. Addison Wesley, 1985.
[15] Aho A. and Ullman, J. D. The Theory of Parsing, Translation and Compiling. Prentice-Hall, 1972.
[16] Gries, D. Compiler Construction for Digital Computers. John Wiley & Sons, 1971.
[17] Fisher, C. N. and LeBlanc, R. J. Crafting a Compiler. Benjamin Cummings, 1988..
[18] Hopcroft, J. D., Ullman, J. D. Introduction to Automata Theory, Languages and Computation. Addison Wesley, 1979.
[19] Harvey, T. J. and Cooper, K. D. Compiler-Controled Memory. In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, October 1998, pp. 2-11, volume 33, number 10. ACM Press, 1998.
[20] Sebesta, R. W., Concepts of Programming Languages,3ª Ed. Addison-Wesley, 1996.