SBL: A Simple Programming Language for Business

Craig E. Ward, Paul Bull
CMSI 585 Programming Languages
Loyola Marymount University

July 4, 2005

1 Introduction

SBL, Simple Business Language, is intended as a straight forward, simple, imperative language for programming business applications. The features of SBL are those that the designers feel are most appropriate for business applications. These include:

- Basic mathematical operations
- Strings as a native type
- Concurrent processing
- Exception handling
- A library of typical accounting functions
- A library of standardized database connectivity routines

SBL should provide programmers with the materials needed to model the steps of a business processes. Such processes are usually, in-and-of themselves, simple and straight forward. The complexities arise from the way each of these steps interact. By allowing a focus on the steps instead of the overall process, SBL should serve the business community better than a language that allows a technical team to attack enterprise issues.

All of the features of SBL can be found in other languages. SBL has been designed from an *a la carte* survey of many programming languages, especially C/C++, Java, Perl, and the much maligned Pascal. The goal is to design a language that allows for a variety of programming styles without allowing so much variety that it becomes easy for the code to be confusing. The *References* section lists the languages surveyed.

2 Syntax and Semantics Overview

2.1 Microsyntax

The source code for a SBL program is a sequence of Unicode characters. Comments start with "//" and extend to the end of line. Language tokens are comprised of the longest possible match of characters. Tokens are separated by white space characters or comments.

Identifiers are single token strings of alphabetic characters, digits, or the underscore character starting with an alphabetic character that are not reserved words. SBL has the following reserved words:

```
boolean catch char do else false float int if null record
secret string synchronized task true try until use void
while
```

Integer literals are a sequence of digits. Floating point literals are identified as a sequence of digits followed by a period (".") followed by at least one digit. Scientific notation for floating point numbers is not part of SBL.[1]

String literals are a sequence of printable characters enclosed between double quotes. The following sequences can be used to encode additional, special characters:

`\n`	Newline
`\t`	Tab
`\"`	Double quote
`\'`	Single quote
`\\`	Back slash
`\xNNNN`	Hexadecimal code point for character NNNN (the number can be any arbitrary number that is a valid code point)

[1] The language also lacks bit operators; SBL should not be used for system programming.

2.2 Macrosyntax

Below is the EBNF description of SBL syntax. Brackets indicate an optional item; tokens in curly braces appear zero or more times. Tokens in lowercase are reserved words.

```
PROGRAM ::== MODULE
MODULE ::== {use ID ';'} {FUNCTION}
FUNCTION ::== [secret] PROFILE BODY
BODY ::== '{' [TASKBLK] SERIALBLK '}'
PROFILE ::== (TYPE | void) ID '(' PARAMS ')'
TYPE ::== (boolean|int|float|char|string|RECORD|ID) { '[' ']' }
RECORD ::== record '{' TYPE ID ';' {',' TYPE ID ';'} '}'
PARAMS ::== [TYPE ['&'] ID {, TYPE ['&'] ID}
TASKBLK ::== {task '{' FUNCTION '}'}
SERIALBLK ::== { STATEMENT | '{' SERIALBLK '}' } | try '{' SERIALBLK
'}' catch '{' SERIALBLK '}'
STATEMENT ::== TYPE ID ':=' EXP ';'
      | ID ':=' EXP ';'
      | EXP ';'
      | if '(' EXP ') '{' SERIALBLK '}' [else if '{' SERIALBLK '}'] [
else '{' SERIALBLK '}']
      | return [EXP] ';'
      | while '(' EXP ')' '{' SERIALBLK '}'
      | do '{' SERIALBLK '}' until '(' EXP ')' ';'
EXP ::== EXP0 { '||' EXP0 }
EXP0 ::== EXP1 { '&&' EXP1 }
EXP1 ::== EXP2 { RELOP EXP2 }
EXP2 ::== EXP3 { ADDOP EXP3 }
EXP3 ::== EXP4 { MULOP EXP4 }
EXP4 ::== [PREFIXOP] EXP5
EXP5 ::== LITERAL | ID | ID '(' ARGS ')' | '(' EXP ')'
LITERAL ::== null|true|false|INTLIT|FLOATLIT|CHARLIT|STRINGLIT
ARGS ::== [ EXP { ',' EXP } ]
ADDOP ::== '+' | '-'
MULOP ::== '*' | '/' | '%'
PREFIXOP ::== '-' | '!'
```

2.3 Semantics

An SBL program is a set of one or more functions. Below is an outline of the semantics of the language.

2.3.1 Abstract Syntax

The abstract syntax tree makes our defining of semantics easier by allowing us to use abbreviated aliases for terms that we defined in our syntax. This makes describing the semantic functions of the language more compact and easier to read.

```
op:   PREFIXOP|ADDOP|MULOP
lit: LITERAL -> null | true | false | INTLIT | FLOATLIT | CHARLIT |
STRINGLIT
id:   ID
e:  EXP -> e'||'e | e && e | e+e| e-e| e*e | e/e | e%e
s:   STMT
sb:  SERIALBLK
tb: TASKBLK
parms: PARAMS
type: TYPE
bod:   BODY
pro:  PROFILE
fun:  FUNCTION
mod:  MODULE
prog:PROGRAM
```

2.3.2 Semantic Domains

The domains of a program consist of modules which carry out behavior that effects the state of a program, files that may be written to or read from and the store which is a mapping of variables to integer values.

```
_: module = int list
_:  file = int list
_: store = ID _ int
_: state = store x {modules} x {files}
```

2.3.3 Semantic Functions

A program is a modification from state to state, state to file, file to state or file to file. As mentioned before a program is comprised of one or more modules. All computation starts and ends with the *begin* function which executes statements and calls other functions. The execution of a statement or sequence of statements changes the state of

computation. The execution of expression or assignments changes the store which is a mapping of IDs to integer values.

```
P: PROGRAM _ MODULE _ {MODULE}
M: MODULE _ state _ state _ {files}
F: FUNCTION _ state _ state
S: STATEMENT _ store _ store
E: EXP      _ store _ LITERAL
SB:SERIALBLOCK _ store _ store

P _ | _ = int(M mod, {M mod})
M _ =   _(fun)+
F (e1,,,,en) _ = E e (S s, F fun (_x.undef))
SB(s1…..sn) _ = S s1 _ …S sn _
S( id := e) _ = _ [E(e)|id]
S(e) _  =  int[E(e)]
S(if e sb1 else sb2) = SB sb1 _ | SB sb2 _
S(if e sb1 elsif sb2 else sb3) = S s1 _ | S s2 _ | S s3 _
S(return e) _ = _ [E(e)]
S(while e sb) _ = E e _ _ S(while e sb)(Sb sb _) | _
S(do sb while e) _ =  SB sb _ E e _(S(do sb while e) | _)
E(true) _ = lit
E(false) = lit
E(e1<e2) _ = E e1 _ < E e2 _
E(e1<=e2) _ = E e1 _  <= E e2 _
E(e1>e2) _ = E e1 _ > E e2 _
E(e1>=e2) _ = E e1 _  >= E e2 _
E(e1=e2) _ = E e1 _ = E e2 _
E(e1!= e2) _ = E e1 _  ≠  E e2 _
E(e1 && e2) _ = E e1 _ _ E e2 _
E(e1 || e2) _ = E e1 _ V E e2 _
E(-e) _ = -(E e _)
E(e1 + e2) _ = E e1 _ + E e2 _
E(e1 - e2) _ = E e1 _ - E e2 _
E(e1 * e2) _ = E e1 _ * E e2 _
E(e1 / e2) _ = E e1 _ / E e2 _
E(e1 % e2) _ = E e1 _ % E e2 _
```

3 Names, Declarations, and Scopes

The initial function is called *begin*, which takes two arguments from the invoking environment: an array of string arguments and an integer count of arguments in the array. The profile of this function is:

```
int begin(string [] arguments, int count)
```

The names used for the formal parameters here are examples only. An SBL program may use any names; it is the types that are required.

Multiple functions may be declared in a file. Unless specifically indicated by the *secret* reserved word, functions are globally visible. No relation exists between the name of the object, *e.g.* file, containing the function and the function itself.

Objects may be declared at any point in a function body, but may not be used until after declared. [2] Objects have lexical scope from the point of declaration to the end of the compassing SERIALBLK. A name may not be reused in a nested SERIALBLK construct.

4 Types

SBL provides a set of types most useful to business data processing as well as some generally useful for programming.

- The type *int* for integers, the range being implementation dependent.
- The type *float* for real, floating point, numbers. An implementation only needs to be able to support a range sufficient to represent $1/1000^{th}$ of a unit value.
- The type *string* representing alphanumeric sequences
- A *record* is a named collection of other types.
- An *array* is a named sequence of a single type.
- A *boolean* may only have a value of *true* or *false*.
- The *char* type may hold a single Unicode character .
- Functions are the base type of SBL programs.

All types are first class types. An object of any type may be passed as an argument to any function and returned from any function. By default, the types *int*, *float*, and *boolean* are passed by value. The types *string*, *record*, *array*, and *function* are passed by

[2] In SBL, object is used to refer to anything that can be named in the language, not as the term is used in object oriented systems.

reference. Pass by value may be overridden by use of the “&” character in the list of formal parameters to a function.

5 Expressions and Statements

Statements produce side effects. An expression produces a value for one of the SBL types. Expressions in actual parameters are evaluated using lazy evaluation. An expression in a simple statement will short circuit for boolean values.

5.1 Examples

- int index := 0;

  The identifier *index* is declared and then initialized with the integer value 0.

- string item_description;

  The identifier *item_description* is declared. The value of the object is undefined.

- overdrawn := true;

  A previously declared boolean, *overdrawn*, is assigned the value *true*;

- if (count <= 100) { order_items(product); }

  The integer value of *count* is compared to 100; if the expression evaluates to *true*, call the function *order_items* with the argument *product*, otherwise do nothing.

- while (next_account != null) {overdrawn = check_acct_status(next_account);

  }

  Until the value of *next_account* becomes null, call function *check_acct_status* with *next_account* as an argument. Assign the returned value to *overdrawn*. The type of next_account must match the expected type of the *check_acct_status* formal parameter and the function’s return type must match the type of *overdrawn*.

6 Subroutines and Modules

6.1 Functions

A function is defined by supplying the keyword ***secret*** if the function is only to remain accessible within its own file, if the ***secret*** keyword is omitted, the function is accessible to other files that may be linked to the functions module. Next the return type is specified for the return value of the function or ***void*** if the function does not return a value. Return types include ***boolean, float, int, char, string, record***, or a user-defined type. Next the name of the function is supplied. After that, the parameters of the function are listed within parenthesis. Parameters are specified by type and then name followed by a comma if an additional parameter follows. Lastly the body of the function is supplied within curly braces. If a return type was specified for the function a return statement returning the appropriate type must be specified at the end of the function or at the end of each block of a sequence of conditionals where appropriate. The general skeleton for function definition is as follows:

```
["secret"] return type | "void" function-name '(' param1,.....paramN')'
'{'
 body
 [return statement]
'}'
```

Examples are as follows:

1.)A function only accessible within its own module:

```
…
secret void modifyFields (int x, char y, float z)
{
  intfield = x;
  charfield = y;
  floatfield = z;
…
}
```

Where *intfield*, *charfield*, and *floatfield* would be variables of the function that called modifyFields or global variables of the module modifyFields is defined in.

2.) A function that is externally accessible and has a return type

```
float addFloats( float x, float y)
{
  return x + y;
}
```

6.2 Modules

6.2.1 Module Creation

A module is defined by creating a file with a list of function definitions in it. The module file is begun with a label within tags that identifies the name of the module and optionally the directory in which it is to be stored in. The file is saved with the *.mod* suffix. The basic structure is as follows:

‘<’[*directoryname*] [‘<=’] *modname*’>’
…
fun1( ){…}
fun2( ){…}
….
funN( ){…}

If the module name is specified with no directory name then it is assumed that it will be stored in the same directory as any file that might link to it. Otherwise, acceptable directory names would be, for example, ***fio*** for the directory of the text input output library or the names of any other libraries of the standard library. If a directory name is supplied that does not exist in the standard library a new directory with that name is created within the standard library.

Examples of module definition are as follows:

1.) An example of a module that is only accessible to programs within its own directory.

```
<mymodule>
    secret void fun1 ( int x , int y) {…}
            float fun2( float z, float v){…}
    ….
            void funN( ){…}
```

2.) An example of a module that would be stored in the ***fio*** library

```
<fio <= txtmanip>
    string  rotateTxt(string x){…}
    string  clipTxt(string x, int begin, int end){…}
    …
```

3.) Lastly we have an example of a module that will be stored in a newly created subdirectory of the standard library.

```
<transactions <= transmod>
    void updateTransaction (record trans){…}
    void deleteTransaction(record trans){…}
    void modTransaction(record trans){…}
    …
```

transactions is not a library included within the standard library but would now be included with file **transmod.mod** containing the updateTransaction, deleteTransaction, modTransaction etc. functions available through the newly created ***transactions*** library.

6.2.2 Utilizing Modules

In order for a program to utilize functions of a particular module, the intent to use that module must be declared somewhere before the functions are defined in that program. The module is utilized by supplying the ***use*** keyword. If the module file is stored within one of the libraries of the standard library then the name of that library following ***use*** will suffice. If the module is a stand-alone module then the name of the module file in tags must follow ***use.*** Also, as mentioned earlier, in this case the module file and the file of the program using it must be in the same directory. Referring to modules in examples 1 and 2 of section 6.2.1 above, here are examples of how to use modules.

1.) An example using the mymodule.mod file used earlier:

```
…
use <mymodule.mod>;
…
…
```

```
begin( )
{….
   fun2(10.2,99.99);
}
```

Once more, it is important to remember that mymodule must be in the same directory as this program's file.

2.)The following is an example of a program that wants to call functions in txtmanip.mod file.

```
…
use <fio>;
…
begin( )
{…
     rotateText("racecar");
  …
}
```

In this case since txtmanip.mod is in the ***fio*** library so only ***fio*** needed to follow ***use.***

7 Concurrency

7.1 Threads

SBL adopts a style of concurrent processing similar to Ada.

A function declaration may include one or more TASKBLKs, each defining a thread of execution. The TASKBLK contains one or more functions. When the SERIALBLK, of the function begins to execute, the initial function of each TASKBLK also begins. The initial function of a TASKBLK may wait for arguments to be passed to it from the SERIALBLK. A TASKBLK function expecting arguments may be called from the SERIALBLK. Calling a TASKBLK function that does not expect arguments is a compiler error.

7.2 Synchronization

Objects shared between concurrently processing tasks can be enclosed in a *synchronized* block to coordinate access similar to the way Java handles the issue. For example:

```
synchronized { int global_counter; }
```

The SBL implementation must provide coordinated access to the variable *global_counter* using the available resources of the host environment.

8 Standard Library

SBL implementations provide a basic set of library calls. Library routines are included in a module by using the use statement. The use statement may appear at any point in a module. For example:

```
use <fio>; // include the standard file i/o routines
```

8.1 FIO: File Input-Output

The File Input-Output library provides routines for basic text and binary input and output. A record is defined that contains the data needed by the host implementation for controlling the location of reads or writes in a file. Some function must be called within a try/catch construct.

Include this line to use this library: `use <fio>;`

Functions defined in this library are:

`record FILE_HDL &create_file(string path);`	Create a new file using the path and return a file handle for it.
`record FILE_HDL &create_filex(string path);`	Create a new file using the path and return a file handle for it and throw an exception on any i/o error.
`boolean error_condition(record &handle);`	Returns *true* if an error condition exists on the handle, else *false*.
`int print(string buffer);`	Print the string to the screen returning the count of bytes written.

`int printline(string buffer);`	Same as print with a forced newline.
`int read(record &handle, string &buffer, int maxcount);`	Read from the file into the buffer up to maxcount. Returns the count of bytes read.
`int readline(string &buffer, int maxcount);`	Read a string from the controlling terminal up to maxcount characters.
`int write(record &handle, record rec);`	Write the binary bit pattern of record *rec* to the indicated file. Return the count of bytes written.
`int write(record &handle, string buffer);`	Write the buffer to the indicated file. Return the count of bytes written.
`int writex(record &handle, record rec);`	Write the binary bit pattern of record *rec* to the indicated file. Return the count of bytes written and throw an exception on any i/o error.
`int writex(record &handle, string buffer);`	Write the buffer to the indicated file. Return the count of bytes written and throw an exception on any i/o error.
`record FILE_HDL &openfile_r(string path);`	Open a file for reading and return a reference to a FILE_HDL record.
`record FILE_HDL &openfile_rw(string path);`	Open a file for random read-write access and return a reference to a FILE_HDL record.
`record FILE_HDL &openfile_w(string path);`	Open a file for writing and return a reference to a FILE_HDL record.
`record FILE_HDL &openfile_rx(string path);`	Open a file for reading and return a reference to a FILE_HDL record and throw an exception on any i/o error.
`record FILE_HDL &openfile_rwx(string path);`	Open a file for random read-write access and return a reference to a FILE_HDL record and throw an exception on any i/o error.
`record FILE_HDL &openfile_wx(string path);`	Open a file for writing and return a reference to a FILE_HDL record and throw an exception on any i/o error.

8.2 NIO: Network Input/Output

The Network Input/Output library provides some basic routines for using network communication resources. The details are implementation dependent. These routines must be used within a try/catch block. NIO handles are compatible with FIO handles. FIO handles can not be used with these routines.

Include this library with: `use <nio>;`

`record NET_HDL &create_connection(string destination, string protocol);`	Open a connection to the selected destination using the selected protocol.
`boolean close_connection(record &handle);`	Close the connection represented by *handle*.
`int send(record &handle, record &buffer, int bytes)`	Send a buffer of records over a connection.
`int receive(record &handle, record &buffer, int bufsize);`	Read a buffer of records from a connection.

8.3 Currency

The currency library provides some basic math routines to aid in the processing of dollars and cents. Include this library module with: `use <currency>;`

`int make_cents(float value);`	Return the float value as an integer X 100.
`float make_float(int value);`	Divide the integer value by 100 and return as a float.
`int add_currency_cents(float val1, float val2);`	Add the two values and return the result as an integer X 100.
`float add_currency_float(float val1, float val2);`	Add two currency values factoring in any rounding errors, if any, of the host environment.
`int subtract_currency_cents(float val1, float val2);`	Subtract the two values and return the result as an integer X 100.
`float subtract_currency_float(float val1, float val2);`	Subtract two currency values factoring in any rounding errors, if any, of the host environment.

8.4 SDBC: SBL Database Connectivity

The SDBC library provides a standard set of functions for interacting with database management systems. Include this library with: `use <sdbc>;`

These functions must be called within a try/catch block.

`record DB_CONNECTION & get_connection(string path);`	Create a connection to a database using the data supplied in path. Returns a reference to a record that may be used with the other calls in the library.
`boolean close_connection(record &connection);`	Close a connection to a database and release any freed resources.
`int execute_query(record &connection, string statement, record &buffer);`	Execute a statement on a connection that is expected to return a set of records.
`string &execute_statement(record &connection, string statement);`	Execute a statement that does not return a set of records. The results are returned as a string.
`int execute_write(record &connection, string statement, record &buffer);`	Execute a statement over the records in buffer.

8.5 Utility

The utility library provides routines that may be of general use in an application. Use this library with: `use <utility>;`

`int now();`	Returns an integer representing the current time.
`string format_time(string format, int time);`	Returns a string representing the textual representation of the time indicated by the integer. The format string is as in the ANSI C `strftime` function.

`int stringtotime(string timestr, string format);`	Returns integer representation of a string representation of a date/time using the format string to parse the date/time string.
`record calendar &make_calendar(int time);`	Return a reference to a record representing the time represented by the integer time. The record is the same as the ANSI C `struct tm` structure.
`int maketime(record &calendar);`	Return the integer representation of the time represented by the calendar record.
`void sort(record &array, int record_size, int length, boolean function compare);`	Sort an array of *length* items of records of size *record_size* using a comparison function that takes two arguments of the record type and returns *true* if the first argument is less than the second, otherwise *false*.
`int sizeofrecord(record r);`	Returns the size in bytes of the argument.

Appendix A

An example of code using synchronization and task blocks:

```
…
synchronized { record account{
                        string firstname;
                        string lastname;
                        int   customer_id;
                        float account_balance
                        }
                }
begin( ){ doTransactions( account);}
void addtoAccount(record account,float amount)
{
   account.balance += amount;
       }
void printAccountBal(record account)
{
    print(“The balance to the account is now” +
account.balance);
}

     void doTransactions( record account)
     {
         int  i = 0;
         float amt = 2.0
         while( i != 1000)
         {
      task{ addtoAccount(account, amt);}
      task{printAccountBal(account);}
          }
}
```

References

Jensen, Kathleen, Niklaus Wirth. *Pascal User Manual and Report, 2nd Edition*. New York: Springer-Verlag, 1974.

Kernighan, Brian W., Dennis M. Ritchie. *The C Programming Language, 2nd Edition.* Englewood Cliffs, New Jersey: Prentice Hall, 1988.

Lischner, Ray. *C++ in a Nutshell: A Language & Library Reference.* Sebastopol, California: O'Reilly, 2003.

Mak, Ronald. *Writing Compilers and Interpreters.* New York: John Wiley & Sons Inc., 1996.

Muchnick, Steven S. *Advanced Compiler Design and Implementation.* San Diego: Academic Press, 1997.

Reese, George. *Database Programming with JDBC and Java, 2nd Edition*. Sebastopol, California: O'Reilly, 2000.

Scott, Michael L. *Programming Language Pragmatics*. New York: Morgan Kaufmann Publishers, 2000.

Stevens, Al. *Wiley's Teach Yourself C++, 7th Edition.* New York: Wiley Publishing, Inc., 2003.

Toal, Raymond J. "The Language Jax." http://www.technocage.com/~ray/notespage.jsp?pageName=jax&pageTitle=The+Language+Jax; accessed 13 November 2003.

Toal, Raymond J. "Introduction to Ada." http://www.technocage.com/~ray/notespage.jsp?pageName=introada&Title=Introduction+to+Ada; accessed 7 October 2003.

Toal, Raymond J. "Introduction to ML." http://www.technocage.com/~ray/notespage.jsp?pageName=introml&Title=Introduction+to+ML; accessed 9 November 2003.

Toal, Raymond J. "Introduction to Perl." http://www.technocage.com/~ray/notespage.jsp?pageName=introperl&Title=Introduction+to+Perl; accessed 11 November 2003.

Wall, Larry, Tom Christiansen, Jon Orwant. *Programming Perl, 3rd Edition*. Sebastopol, California: O'Reilly, 2000.