1994

SBL- The Structural Biology Language

Marius A. Cornea-Hasegan

Dan C. Marinescu

Report Number:
94-008

SBL - THE STRUCTURAL BIOLOGY LANGUAGE

Marius A. Cornea-Hasegan
Dan C. Marinescu

CSD-TR-94-008
February 1994

SBL - The Structural Biology Language *

Marius A. Cornea-Hasegan
Dan C. Marinescu

February 6, 1994

Abstract

SB is a problem solving environment for structural biology. At the present time, it consists of programs for Iterative Electron Density Averaging, IEDA, programs for optical data processing, and graphics support. SBL is a problem description language allowing a structural biologist to specify complex execution sequences for programs in SB. Currently, it supports execution of programs from the IEDA suite, on distributed memory MIMD systems.

Contents

*This research is supported in part by the NSF under grants CCR-9119388 and BIR-9301210

1 The Structural Biology Language (SBL)

The Structural Biology Language allows a simple description of an execution sequence for programs in the SB problem solving environment. SBL is a Problem Specification Language, limited to a set of language constructs that allow structured programming, and place it closer to the high-level Algol-like languages.

Two categories of programs from the SB environment can be invoked from an SBL program:

(a) **processing programs**, like Envelope, FFTinv, Recip, FFTexp, FFTsynth, and Rotation. More programs will be included as the SB environment evolves.

(b) **auxiliary programs**, designed to support the appropriate sequencing of the programs in the first category, to test some of their output, and to transform part of the control input data for these programs. At the present time, this category includes the programs Converged, Phaseext1, and Phaseext2. A description of these programs follows.

The SBL also includes a minimal set of data types, operations and sequencing constructs that allow a concise description of the processing flow. The possibility of executing each of the SB programs from the command-line is still kept intact.

The main advantages of using the SBL for specifying the execution of SB programs are:

- a given processing sequence can be specified in a very simple way; the user is relieved of the task of knowing the commands necessary to run it on a given machine (several target machines will be supported, including the iPSC/860, the Paragon, and also Sun, IBM RS6000, and SGI workstations; currently, only the iPSC/860 and the Paragon versions are operational).

- multiple nested iterations, and two different levels of phase extension used for Iterative Electron Density Averaging, IEDA, can be executed without user intervention; the user can examine partial results, but does not have to specifically initiate each computational step.

- the user can stop the computation at any time, and then restart it; the same holds true for the case when the computational process was interrupted, e.g. because of a hardware failure, or because the allocated running time was exhausted: it can be resumed at any time later, without any other loss than possibly re-running part of one SB program.

- the user can backup a number of computational steps (up to one innermost cycle for iterative computations), by editing a log file that is generated by the compiled SBL program (see §6, 'Internals of the SBL Programs Execution').

- the execution of a sequence of SB programs will stop as soon as one of them fails, insuring data integrity in this way; the user can then determine and remove the cause of the failure, and restart execution as if nothing has happened.

Every SBL program is compiled into a UNIX shell script (UNIX command list), which is able to control execution for the sequence of IEDA programs.

Compilation of an SBL program can be done either from the UNIX shell level, using the command-line version of the compiler, or from the SB User Interface, as described in a subsequent section.

2 The Structure of an SBL Program

An SBL program consists of declarations, followed by executable statements. The declaration section can be omitted if no variables are used, but the statements part is compulsory. Moreover, there has to be at least one call to an SB program (`envelope`, `fftinv`, `recip`, `fftexp`, or `fftsynth`) for a successful compilation. Note the equivalence between the names of the SB processing and auxiliary programs supported in the language, and their SBL counterparts:

SB Program	SBL Name
Envelope	envelope
FFTinv	fftinv
Recip	recip
FFTexp	fftexp
FFTsynth	fftsynth
Rotation	rotation
converge	converged
phaseext1	phaseext1
phaseext2	phaseext2

2.1 Symbols and Separators

Identifiers in SBL consist of letters, digits, and the underscore character. The first character has to be a letter, and the last character cannot be an underscore (_), these cases being reserved for special identifiers defined by the SBL compiler.

The separator used for declarations or statements, is the semicolon (;). The colon (:) is used to separate a variable list and the keyword specifying the data type in a variable declaration. The comma (,) is used to separate multiple variables in a variable declaration, or the arguments to a procedure or function (only built-in procedures and functions are provided).

2.2 Comments

Comments in SBL are strings of characters delimited by '/*', and '*/'. For example:

```
/* This is a comment */

/*
       This is another comment
*/
```

2.3 Data Types and Declarations in SBL

Four data types are supported: integer, real, string, and boolean. User defined data types are not allowed. The following subsections on data types also illustrate the use of the declarations in SBL: a variable name, or a sequence of variable names separated by commas, are followed by a colon, and a keyword designating the data type.

The SBL is a strongly typed language, and it allows only safe expressions.

All variables declared in an SBL program are automatically initialized to a default value: 0 for integers, 0.0 for reals, `false` for boolean variables, and the empty string "" for string variables.

Integer

The variables of type integer are declared using the keyword `integer`. In the following examples, i, j, int_var, and I are declared as integer variables:

```
i, j, int_var : integer;
I : integer;
```

Arithmetic and relational operations are possible on variables of type integer, that can also be mixed with integer constants.

String

String variables are provided only as a convenience, e.g. when long path names are to be passed as arguments to the SB procedures and functions. Only one assignment is allowed to each string variable in the program. String constants are double-quoted strings of characters (any characters except double-quotes). Examples of string variable declarations are:

```
outfile : string;
envexec0, envexec, envcntlin8, envcntlin9, envcntlin89 : string;
```

String constants are of type string, and can be mixed with string variables. The only operations that can be performed on strings, are testing for equality, or non-equality.

Boolean

Boolean variables are provided to retain the values returned by the boolean functions of the SBL. Conditional expressions can then be built using also the boolean constants `true` and `false`. The operations that can be applied, are the logical operations, the equality, and the non-equality tests. The following are boolean variables:

```
hasconv, ext1poss : boolean;
```

Real

The real type variables are declared using the keyword `real`:

```
incr, delta : real;
```

The real constants are also of type real. The real variables and constants are mainly used to be passed as arguments to the SB equivalents of the built-in functions. The only operations that can be performed on real quantities, are the equality, and non-equality tests.

2.4 Constants

Constants of the types provided for variables, are supported by the SBL. Some examples are:

```
- constants of type integer: 3, 33, -25, 0
- constants of type string:

        "/home/ipscgate/u10/userx/cycle/fftinv.in",
        "/home/ipscgate/cryst/Prog/i860/Envelope/V1/1.2/Envelope",
        "my_own constant, quite peculiar"

- constants of type boolean: true, false
- constants of type real: 3.14, 0., .0, 0.0001
```

2.5 Built-in Procedures

The built-in procedures are the SBL correspondents of the processing programs recognized by SB: `envelope()`, `fftinv()`, `recip()`, `fftexp()`, `fftsynth()`, *and* `rotation()`.

All the built-in procedures take three arguments of type string, i.e. string variables, or string constants. The three strings represent, in order, the paths for the executable program, the control input file, and the control output file of the respective program.

The built-in procedures cannot be mixed in any kind of expression, or assignment statement. The parentheses are required to follow the built-in procedure name, even if no arguments are used.

The arguments to a built-in procedure are all of type 'value'. For example, the following declarations and call illustrate the invocation of the `envelope()` procedure:

```
outfile : string;
envexec0, envcntlin8 : string;
```

```
...

/* assign values to envexec0, envcntlin8, outfile     */

...

envelope(envexec0, envcntlin8, outfile);
```

2.6 Built-in Functions

The built-in functions are all of type boolean, and correspond to auxiliary SB procedures, meant to help in the proper sequencing of the SB processing programs. As for the built-in procedures, the built-in functions use the 'call-by-value' mechanism.

Currently, three built-in boolean functions used by the iterative electron density averaging are available: `converged()`, `phaseext1()`, and `phaseext2()`. None of these functions can appear in boolean expressions. The values returned have to be assigned first to boolean variables, that can be used then in boolean expressions, in combination with other boolean constants, variables, or expressions.

The function **converged()** takes one real argument, variable, or constant, that is the increment for which the overall correlation coefficients have to be tested after a call to `recip()`, in order to determine if convergence has been reached in the iterative electron density averaging process at a given resolution.

The function returns `true` if convergence was reached, and `false`, otherwise.

For example, if `delta` is a real variable of value 0.005, then the call

```
converged(delta)
```

returns `true` if the values of the overall correlation coefficients in the last two calls to `recip()`, differ by less than 0.005 (convergence was reached), and it returns `false` otherwise.

The **phaseext1()** function performs phase extension after convergence has been reached at a given resolution, but maintaining the grid size and the mask file. A new value is computed for the upper resolution of the structure factors, and is written out to the control input files for the FFTinv, Recip, and FFTexp programs. New values are also computed for the structure factor indices, `hmax`, `kmax`, and `lmax`, and are written out to the control input file for the FFTinv program.

The function takes four arguments: three arguments of type string, representing, in order, the paths for the control input files for FFTinv, Recip, and FFTexp programs, and one real argument (constant, or variable), that represents the increment used in computing the new values of `hmax`, `kmax`, and `lmax`. Assuming for example that `incr` is the real variable (having e.g. the value 1.0, assigned to it previously), and that `invcntlin`, `reccntlin`, and `expcntlin` are the three string variables, then `phaseext1()` is invoked as:

```
phaseext1(invcntlin, reccntlin, expcntlin, incr)
```

The function returns `true` if phase extension is possible at the given grid size, and `false`, otherwise. If phase extension is possible, then as a side effect, the control input files are modified.

As for the `converged()` function, the value returned by `phaseext1()` has to be assigned to a boolean variable before being tested or used in a boolean expression of any kind (see the section on the 'Assignment Statement' below).

The function **phaseext2()**, is provided to perform phase extension when the current grid size does not allow the use of `phaseext1()` anymore. This has to be combined with the generation of a new mask, too. This function is not operational as of Jan 12, 1994, in that it has no equivalent in the set of SB support programs. From the SBL point of view, though, it is equivalent to the `phaseext1()` function, which will allow an easy implementation of the operational `phaseext2()`.

For more information about these auxiliary procedures, see §9, the section on 'Auxiliary SB Procedures'.

2.7 Input/Output

There is only one I/O function in SBL, `print()`, that writes to the standard output. There is no function to read from the standard input, or to write to the standard error output. All input data has to be provided through assignment statements.

The `print()` function takes any number of arguments, variables, or constants, of any type. At execution, the constants are printed as they are at the standard output. For variables, their values are printed. A newline character is automatically appended at the end. For example, assuming that p is a real variable having the value 3.14, and i is an integer of value 25, the call:

```
print("The value of i is ", i, " and of p is ", p,
        " ; p is larger than ", 3);
```

will print:

```
'The value of i is 25 and of p is 3.14 ; p is larger than 3'
```

The `print` function has to appear in a stand-alone statement, as in the example above.

2.8 Operators

The operators supported in SBL are arithmetic, relational, and logical.

Arithmetic Operators

The binary arithmetic operators are plus (+), minus (-), times ($\star$), div (/), and modulus (%), with the obvious meanings. The minus (-) is also a unary operator. The arithmetic operators can only be applied to integer variables and constants. For example, if i and j are integer variables, then the following is a valid arithmetic expression:

```
((-i) % 7) + ((j * 5) / (-4) - 3)
```

Relational Operators

The equality operator (=), and the non-equality one (<>) are binary operators that can be applied to any two arguments of the same type (integer, real, string, or boolean). The result is a boolean expression.

See the section on 'Operators, Precedence, and Associativity' below, in order to decide if these expressions need to be parenthesized, depending on the context of their use.

The other relational operators are also binary, but can only be applied to integer operands. These are 'less than' (<), 'greater than' (>), 'less than or equal to' ($\leq$), and 'greater than or equal to' ($\geq$).

Logical Operators

The logical operators are and, or (binary), and not (unary). They can be applied only to boolean operands (constants, variables, or expressions), and the resulting expression is of type boolean.

Operators, Precedence, and Associativity

A summary of the SBL operators, their type, the type(s) of their argument(s), the type of the result, and the operator's precedence, is:

Operand	Type	Argument(s)	Result	Precedence
-	unary	integer	integer	4
$\star$, /, %	binary	integer	integer	4
+, -	binary	integer	integer	3
<, >, $\leq$, $\geq$	binary	integer	boolean	2
=, <>	binary	integer, real, boolean, string	boolean	2
not	unary	boolean	boolean	1
and, or	binary	boolean	boolean	0

Some of the operators are non-associative, and therefore need to be fully parenthesized when used in combination with other operators of the same precedence, while other operators are left-associative. The non-associative operators are the logical `not`, the relational '=', '<>', '<', '≤', '>', and '≥', and the arithmetic unary '-'. The left associative operators are the binary arithmetic '+', '-', '*', '/', and '%', and the logical `and` and `or`.

For example, if a and b are boolean variables, then the expression:

a and b or not a and not b

is equivalent to the following expression:

((a and b) or (not a)) and (not b)

To have the expression evaluated as it would be in languages where the precedence of the `and` operator is higher than that of the `or` operator, some parentheses are necessary, as shown below:

(a and b) or (not a and not b)

2.9 Statements

A statement is either a simple statement, or a list of statements. Simple statements are terminated by a semicolon (;). These are : the `null` statement, the assignment statement, the procedure call, the `while` statement (the only repetitive statement), and the `if` statement (the only conditional statement).

Null Statement

The `null` statement is used if a branch of an `if` statement does not contain anything else, or if the body of a repetitive statement (`while`) needs to be empty. The `null` statement does not have any effect, and its syntax is:

```
null;
```

Assignment Statement

The assignment statement describes actions to be performed on the SBL program's data. It specifies that a newly computed value (or a constant) is to be assigned to a variable. The left-hand side of an assignment statement contains a variable name. The assignment symbol ':=' follows, and the right-hand side contains either a constant, or an expression built by means of the SBL operators. If the left-hand side represents the name of a variable of type string, then the right-hand side needs to be a constant. This happens because the string data type is provided only to allow 'shorthands', in the form of variable names, for constants represented by long strings of characters. If the left-hand side of an assignment statement contains the name of a variable of type real, then the right hand side can only be a constant

or a variable of type real, as no expressions built with the SBL operators yield values of type real. If the left hand side of an assignment statement is a variable of type boolean, then the right hand side can only be a boolean constant, or a built-in function of type boolean.

Expressions of type boolean are allowed as conditions for the conditional, and for the repetitive statements, while expressions of type string or real cannot be used in SBL programs.

If the variable on the left-hand side is of type integer, then compound expressions are also allowed on the right-hand side. A few examples of assignment statements are following, preceded by the respective variable declarations:

```
i, j, k : integer;
b, c, d : boolean;
p, q : real;
s, t, u : string;

...

i := 16;
j := (((i % 5) - 3) * (i / 3)) / (i - 2) + 6;
k := 1;
p := 0.001;
q := 3.1415926;
b := converged(p);
c := converged(0.0005);
s := "/home/sampson6/cryst/TESTV1.2/fftinv.in";
t := "/home/sampson6/cryst/TESTV1.2/recip.in";
u := "/home/sampson6/cryst/TESTV1.2/fftexp.in";
d := phaseext1(q, s, t, u);
```

Built-in-Procedure Call

A procedure call to a built-in SB procedure like `envelope`, `fftinv`, `recip`, `fftexp`, `fftsynth`, or `rotation`, can only occur in a statement of the form:

```
envelope(execpath, cntlinpath, cntloutpath);
```

In this example, `execpath`, `cntlinpath`, and `cntloutpath` are the paths (in the form of string variables, as assumed here, or in the form of string constants), of the Envelope executable program, of the control input file, and of the control output file. Internally, the procedures are of type void (which does not have to be explicitly declared), and do not match the types of any other SBL variables.

Repetitive Statement

The repetitive statement has the syntax:

```
while (condition) loop

        statement list

end;
```

where 'condition' is a boolean expression, and 'statement list' is a simple statement, or a list of simple statements, which constitutes the body of the `while` loop. The statement list is executed 0 or more times, depending on the logical value of the condition, which is evaluated before entering the body of the loop.

Conditional Statement

A conditional statement has the syntax:

```
if (condition_1) then
        statement-list_1
elsif (condition_2) then
        statement-list_2

...

elsif (condition_n) then
        statement-list_n
else
        statement-list_(n+1)
endif;
```

The expressions $\texttt{condition}_1$, $\texttt{condition}_2$,... are of type boolean (boolean variables in the simplest case).

The 'elsif' parts are optional, as is also the 'else' part, together with their respective associated statements. The final 'endif' cannot be omitted, though.

A single simple statement is recommended per line of SBL source program, with the exception of the `if` and `while` statements, for which the structure has to be the one given in the examples above. This limitation exists because the mechanism used to restart the program if it has not run to completion, associates the line number in the source SBL program with each statement, and uses it when resuming execution.

If this rule is not obeyed, the program will run correctly if not interrupted, but a restart may fail.

3 Sample SBL Programs

Example 1

A simple example follows, in which only one SB processing program is invoked. The following SBL source program was created in a file named 'example':

```
/* Program start                                                    */
/* Declarations */
outfile : string;
envexec, envcntlin89 : string;

/* Statements */
outfile := "all.out";
envexec := "/home/ipscgate/u10/cryst/Prog/i860/Envelope/V1/1.2/ENVELOPE";
envcntlin89 := "/home/ipscgate/u10/cornea/TESTV1.2/env89/envelope.inj_89";
envelope(envexec, envcntlin89, outfile);

/* Program end                                                      */
```

The executable program to be used has the path

```
'/home/ipscgate/u10/cryst/Prog/i860/Envelope/V1/1.2/ENVELOPE',
```

the control input file is

```
'/home/ipscgate/u10/cornea/TESTV1.2/env89/envelope.inj_89'
```

and its name suggests that it is set up for running the options 8 and 9 of the Envelope program, while the control output file has the name 'all.out', and will be generated in the current directory. In the same directory, a log file, named 'log.out' will be created.

The compiled version of this program will be a UNIX shell script which, when invoked, will allocate the specified partition to run the Envelope program, and will start its execution. The log file will contain information about the run, used also by the restart mechanism, useful especially when more SB programs are run consecutively.

Example 2

This is a more complex example, and achieves the following tasks: runs options 8 and 9 of the Envelope program as two different jobs, then executes two nested `while` loops. The inner loop runs the FFTinv program, the Recip program, and then tests for convergence (by checking on the overall correlation coefficients from the last two runs of Recip). If convergence was not achieved, the FFTexp program, the FFTsynth program, and the Envelope program (options 8 and 9 as one single run), are executed next. If a certain maximum number of iterations specified for the inner loop was reached, or if convergence was achieved, the inner

loop is left, and if phase extension at the present grid size is possible, it is performed through a call to phaseext1() (see §2.6 on 'Built-in Functions' for details on phaseext1()). The inner loop is then resumed. Both loops contain print() statements for informative messages.

The SBL source program, created say in a file named 'cycle', follows:

```
/* Program start                                                    */

/*      Averaging cycle - test for the UI and the SBL compiler      */
        /*      Jan 12, 1994                                        */

        delta : real;
        numiter : integer;
        maxiter : integer;
        phase_ext1_cnt : integer;
        hasconv, ext1poss, exit : boolean;
        outfile : string;
        envexec0, envexec, envcntlin8, envcntlin9, envcntlin89 : string;
        invexec, invcntlin : string;
        recexec, reccntlin : string;
        expexec, expcntlin : string;
        synthexec, synthcntlin : string;


        delta := 0.005;
        numiter := 0;
        maxiter := 100;
        hasconv := false;
        outfile := "all.out";
        envexec0 := "/home/ipscgate/u10/cryst/Prog/i860/Envelope/V1/1.0/ENVELOPE";
        envcntlin8 := "/home/ipscgate/u10/cornea/TESTV1.2/env8/envelope.inj_8";
        envexec := "/home/ipscgate/u10/cryst/Prog/i860/Envelope/V1/1.2/ENVELOPE";
        envcntlin9 := "/home/ipscgate/u10/cornea/TESTV1.2/env9/envelope.inj_9";
        invexec := "/home/ipscgate/u10/cryst/Prog/i860/FFTinv/V1/1.0/n0/NODE0";
        invcntlin := "/home/ipscgate/u10/cornea/TESTV1.2/fftinv/fftinv.inj";
        recexec := "/home/ipscgate/u10/cryst/Prog/i860/Recip/V1/1.0/NODE0";
        reccntlin := "/home/ipscgate/u10/cornea/TESTV1.2/recip/recip.inj";
        expexec := "/home/ipscgate/u10/cryst/Prog/i860/FFTexp/V1/1.0/NODE0";
        expcntlin := "/home/ipscgate/u10/cornea/TESTV1.2/fftexp/fftexp.inj";
        synthexec := "/home/ipscgate/u10/cryst/Prog/i860/FFTsynth/V1/1.0/NODE0";
        synthcntlin := "/home/ipscgate/u10/cornea/TESTV1.2/fftsynth/fftsynth.inj";
        envcntlin89 := "/home/ipscgate/u10/cornea/TESTV1.2/env89/envelope.inj_89";
```

```
envelope(envexec0, envcntlin8, outfile);
envelope(envexec, envcntlin9, outfile);

ext1poss := true;
exit := false;
phase_ext1_cnt := 1;

while(ext1poss = true and exit = false) loop

        print("CYCLE AT PHASE EXTENSION STEP ", phase_ext1_cnt);

        while(numiter < maxiter and not hasconv) loop

                numiter := numiter + 1;

                fftinv(invexec, invcntlin, outfile);
                recip(recexec, reccntlin, outfile);

                hasconv := converged(delta);

                if(not hasconv) then

                        fftexp(expexec, expcntlin, outfile);
                        fftsynth(synthexec, synthcntlin, outfile);
                        envelope(envexec, envcntlin89, outfile);

                endif;

        end;

        if(numiter = maxiter and not hasconv) then

                print("DID NOT CONVERGE TO DELTA = ", delta, " IN ",
                        maxiter, " ITERATIONS");
               exit := true;

        else

              print("CONVERGED TO DELTA = ", delta, " IN ", numiter,
                        " ITERATIONS");
```

```
                ext1poss :=
                        phaseext1(invcntlin, reccntlin, expcntlin, 1.0);
                phase_ext1_cnt := phase_ext1_cnt + 1;

                hasconv := false;
                numiter := 0;

        endif;

end;

print("TERMINATED PHASE EXTENSION AT PHASE EXTENSION STEP ",
        phase_ext1_cnt);

/* Program end                                                          */
```

4 The SBL Compiler

The most general definition of a compiler considers a set of pairs (x,y), where x is a source language program, and y is a target language program into which x is translated. The set of pairs is assumed to be known beforehand, and the compiler is the device that given x, efficiently translates it into y. The set of pairs (x,y) is referred to as a translation. If x is a string over an alphabet S, and y a string over D, then a translation is merely a mapping from S* to D* [Aho 72].

The source language for the SBL compiler is the Structural Biology Language (SBL) described herein. The target language is the UNIX Shell Programming Language, with particular constructs depending on the target machine.

The SBL compiler can be invoked in its command-line version, or from the SB User Interface.

4.1 The Command Line Version

In the command-line version, the name to use in order to invoke the compiler, is 'sblc' (from 'Structural Biology Language Compiler'). This name has to be followed by the name of the SBL source program to compile. The compiled output is written to a file having the same name as the input, plus the suffix '.e'.

The compiled program is a UNIX shell script, targeted to a specific architecture, and OS. The compiler has several switches used to specify the target machine:

-i : compile for the iPSC/860 Intel hypercube
-p : compile for the Intel Paragon machine
-s : compile for Sun workstations
-r : compile for IBM RS6000 workstations

The default is '-i'. Only the '-i' and '-p' flags are operational as of Jan 12, 1994.

For example:

```
sblc example
```

will result in a UNIX shell script for executing the Envelope program on an iPSC/860, written to the file named 'example.e'

The following command will compile the SBL program 'cycle', for execution on the Intel Paragon supercomputer:

```
sblc -p cycle
```

The output is in the file named 'cycle.e' in this case.

4.2 The SB User Interface version

The SB User Interface version of the compiler is invoked through a series of submenus: 'ITERATIVE ELECTRON DENSITY AVERAGING' from the main menu, then 'SBL Program', and 'SBL Object Program' object selection (see figure 5, on page 49). If there exists an SBL program object in the catalog of objects, that we want to modify, then the object needs to be selected, and the 'Create/Modify SBL Prog' button opens a window, in which it starts a copy of the 'vi' editor, with the SBL source program read in, and ready to modify. If a new SBL Program is to be created, an object of type SBL has to be created first, e.g. using the 'New Object' button. Then, the 'Create/Modify SBL Program' button can be used too to open a window with the 'vi' editor started, but no file read in this time.

In either case of the two above, when the editor is quit, the SBL compiler is automatically invoked to compile the SBL source program that was created or modified. If the compilation is successful, the user has the possibility of viewing the compiled file, of copying it to the (possibly remote) machine on which it will be executed, or to quit. If the compilation is not successful, error messages are displayed, and the user has the option to edit again the file and to recompile, or to quit at this point.

Unlike for the command-line version, the SB User Interface version of the compiler packs the SBL source program and the compiled UNIX shell script into one file, in which the source program appears as a comment at the beginning. When an existing program is to be modified from the User Interface, the SBL program is unpacked from this combined file. For this reason, a UNIX shell script compiled with the command-line version of the compiler, cannot be used with the User Interface to extract the source program from it. In order to see the structure of a packed file, the user can view it from the User Interface, after compilation. The windows mentioned in the description above, are illustrated in §11, 'SB User Interface Support for SBL Program execution'.

5 Running SBL Programs. Stopping and Resuming the Computation

As for the compilation, the user can run a compiled SBL program from either the command line, or from the SB User Interface.

5.1 The Command Line Version

Once an SBL program is compiled, the UNIX shell script that was created, can be transferred to the machine it was generated for, and then executed. The shell script name has to be followed by a partition name, and by the number of nodes in the partition. An ampersand at the end will run the job in the background. For example, the shell script named cycle.e, can be run on a partition of 32 nodes, named 'mypart', by issuing the following command:

```
cycle.e mypart 32 &
```

Running the job in the background leaves the user with the possibility of examining partial results, or running other jobs, too.

If the user wants to stop the execution of a job, then the command to use is:

```
kill %1
```

assuming that this was the first job started in the background.

To start the SBL program from the beginning, the log file, 'log.out', must not exist in the shell script's directory. The output file may exist, but the current output will be appended at its end.

If the job was killed, or has exhausted the allocated running time, or stopped running because of a hardware problem, it can be restarted by typing the command:

```
cycle.e &
```

The partition name and the number of nodes, as well as the point in the SBL program where the execution has to resume, are all recovered from a log file, named 'log.out', created in the directory that contains the UNIX shell script. Restarting the program will fail if the log file does not exist, or if it has been corrupted.

If the user wants the job to be run even after he/she logs out, then a Bourne shell ('sh') has to be started, and the 'nohup' command has to be used. For starting afresh the SBL program, type:

```
% sh
$ nohup csh cycle.e mypart 16 &
$ exit
% logout
```

To restart the program, the same commands are to be used, but without specifying the partition name, and the number of nodes. In either case, the output of the program that would have gone to the screen, goes now to a file named 'nohup.out'.

5.2 The SB User Interface version

To run an SBL program from the SB user interface, the user has to create and compile it first, unless an existing program is used.

To start the execution of a program from the beginning, the following steps are necessary:

- Select the SBL program from the 'SBL Program' object selection window.

- Press 'OK', and the window disappears if the SBL compiled program was found on the target machine.

- Press 'Done' in the 'SBL Program' submenu, and the 'Info for the SBL Program Execution' window appears on the screen; the labels in the left hand column are also help buttons.

- Type in the partition name and the number of nodes in the partition, in the appropriate text editors. Note that the path of the UNIX shell script is also displayed, but it cannot be changed by the user at this point.

- Press 'Run', and a window opens, displaying the output for the program execution, that goes to the screen on the target machine; if the files in the log directory need to be examined, a 'cd' has to be made to the place displayed on the screen.

- If starting program execution fails for a reason that may be corrected, the window that was open on the remote machine may be closed; after correcting the factors that caused execution failure, the 'Run' button can be pressed again.

As for the command-line version, starting a program from the beginning will fail, if the 'log.out' file exists in the directory containing the compiled UNIX shell script.

In order to restart an SBL program that was interrupted for some reason, the same sequence of steps as above has to be followed, except that the partition name and the number of nodes do not have to be specified in this case, and the 'Restart' button has to be pressed instead of the 'Run' button, in the 'Info for the SBL Program Execution' window.

6 Internals of the SBL Program Execution

One of the main advantages of using an SBL program to run a sequence of SB processing programs, is the capability of resuming execution, if it has been interrupted for some reason. This is achieved by having both the SB processing programs (`envelope`, `fftinv`, `recip`, `fftexp`, `fftsynth`, and `rotation`), and the SBL program itself, write messages to a log file.

The SB programs write to the log file only one line, at the completion of their execution. For example, for the Envelope program, this line is:

```
End-of-Envelope
```

When the SBL program has to initiate execution of the SB program following the Envelope program that has written the message above, it checks the log file for the existence of a last line, containing the string 'End-of'. If it is found, execution of the next program is initiated. If not, it means that the previous SB program (Envelope), has not completed normally, and the SBL program stops execution.

The SBL program writes to the log file some information when it completes the initialization phase (see the example below), and then, before initiating execution of each SB processing program, it writes the current date and time, a line indicating which program is going to start, and also the current status of the SBL program. This helps the SBL program 'know' where to restart execution at a later time.

The SBL program status contains the values of all the integer, real, and boolean type variables defined in the SBL program. The variables of type string are not included, as only one assignment is allowed for them. All the assignments to the string variables will be executed at a restart, without any risk.

One other piece of information completes the program status: the 'restart line number'. This is the line number for the line of SBL source program, that contains the procedure call for the SB program which is going to be executed next. As an example, part of the 'log.out' file created by the UNIX shell script named 'cycle.e' is following:

```
Start execution of '/home/ipscgate/u10/cornea/TESTV1.2/gsscycle/cycle.e'
cubename m1615013 numnodes 16

End-of-Initialization

Tue Jan 25 13:25:32 EST 1994
Start of the envelope program
PROGRAM STATUS : _rln  phase_ext1_cnt exit ext1poss hasconv maxiter numiter delta
PROGRAM_STATUS: 38 0 0 0 0 100 0 0.005000
End-of-ENVELOPE

...
...

Tue Jan 25 19:55:51 EST 1994
Start of the fftinv program
PROGRAM STATUS : _rln  phase_ext1_cnt exit ext1poss hasconv maxiter numiter delta
PROGRAM_STATUS: 53 4 0 1 0 100 1 0.005000
```

The line containing the program status (beginning with 'PROGRAM_STATUS'), is preceded by a line containing the names of the variables that are saved. This makes the information user-readable, and can be useful in debugging operations.

In the example above, the last program whose execution has started, is the FFTinv invoked on line 53 in the SBL source program. The value of 'hasconv' is 0, or `false` (convergence was not reached at this step of phase extension), 'numiter' is 1 (the inner `while` loop is executed the first time at this step of phase extension), 'phase_ext1_cnt' is 4 (this is the fourth step of phase extension), and 'ext1poss' is 1 (phase extension at the same grid size is possible; 1 is the equivalent of `true`). The FFTinv program has not completed, as the line containing 'End-of-FFTinv' does not appear above. Assuming that the SBL program execution terminated at this stage of the 'log.out' file, it can be restarted, as indicated before, by simply typing

```
cycle.e &
```

while in the directory containing this UNIX shell script. The 'log.out' file will look as follows after execution has resumed:

```
...
...

Tue Jan 25 19:55:51 EST 1994
Start of the fftinv program
PROGRAM STATUS : _rln  phase_ext1_cnt exit ext1poss hasconv maxiter numiter delta
PROGRAM_STATUS: 53 4 0 1 0 100 1 0.005000
RESTART
Wed Jan 26 18:46:53 EST 1994
End-of-Initialization

Wed Jan 26 18:46:59 EST 1994
Start of the fftinv program
PROGRAM STATUS : _rln  phase_ext1_cnt exit ext1poss hasconv maxiter numiter delta
PROGRAM_STATUS: 53 4 0 1 0 100 1 0.005000
End-of-FFTINV

Wed Jan 26 18:51:05 EST 1994
Start of the recip program
PROGRAM STATUS : _rln  phase_ext1_cnt exit ext1poss hasconv maxiter numiter delta
PROGRAM_STATUS: 54 4 0 1 0 100 1 0.005000
End-of-RECIP

...
...
```

In this example, the FFTinv program that was interrupted completes, then Recip follows, a.s.o.

If the log file is somehow corrupted, the restart may fail. The user has in this case the possibility to edit the 'log.out' file, so that the last line be of the appropriate type. Also, the user can delete a number of lines at the end of this file, which is equivalent to backing up a number of steps in the computation (one 'step' meaning here the execution of one SB processing program).

A special case may occur though. A problem arises if the execution of the SBL program was interrupted by some external cause *after* an SB processing program has completed, but *before* the next one has started. This situation, which is very unlikely to occur (for the iPSC/860 the host machine, and for the Paragon system, the service node would have to fail for this to happen), can be corrected by the user in two possible ways, both involving a modification to the log file ('log.out'). The first possibility is to delete the last line in the log file (the one containing 'End-of-...'), but this would mean re-running the last SB processing program, that completed successfully. The second possibility requires more care from the user: a group of 5 lines can be added (actually, only the last one is necessary), as if the execution of the next SB processing program had started, but not completed:

```
USER ADDED 4 LINES
Tue Jan 26 22:00:00 EST 1994
Start of the fftinv program
PROGRAM STATUS : _rln  phase_ext1_cnt exit ext1poss hasconv maxiter numiter delta
PROGRAM_STATUS: 53 4 0 1 0 100 1 0.005000
```

In this example, we assumed that the FFTinv invoked by the `fftinv()` on line 53 was the next to start, and we have computed and saved the restart line number and the variable values, as if the SBL program had been executed up to line 53 in the SBL source program (the one containing the call to the `fftinv()` built-in procedure).

The atomicity of the SBL program execution is at the level one SB processing program. Each such program uses input data files of which at least some may be created by other SB programs executed previously, and generates usually one data output file. Therefore, when interrupted, and then restarted, an SBL program will re-execute at most part of one SB processing program (and some SBL statements that do not involve the SBL processing programs).

When backing up a number of steps in the computation, the availability of the input data files has to be taken into consideration. For iterative computations, the output data files are overwritten. Hence, unless other special arrangements are made, the user can only back up a number of steps equal to the 'size' of the innermost `while` loop.

7 The SBL Reference Manual

A description of the *context-free grammar, cfg,* defining the SBL follows, in an extended BNF (EBNF) notation.

Recall that a context-free grammar is a restricted grammar, with productions satisfying certain conditions.

The context-free grammar is defined as a quadruple G = (N, Σ P,S), where:

- N is a finite set of non-terminal symbols (also called variables, or syntactic categories)

- Σ is a set of terminal symbols, disjoint from N

- P is a finite subset of

  N X (N U Σ)*

  An element (α, β) in P is written as $\alpha \rightarrow \beta$, and called a production.

- S is a distinguished symbol in N, called the sentence (or start) symbol

The notations used in our EBNF description are:

<x>	non-terminal symbol
x	terminal symbol
[x]	optional symbol, that may occur 0 or 1 times
{x}	symbol repeated 0, 1, 2, ... times
(x|y|z)	exactly one of the symbols x, y, or z must appear

The cfg in EBNF notation follows:

```
<program>      ::= <declarations><stmtlist>

<stmtlist>     ::= <statement>{<statement>}

<statement>    ::= null ;
                   ::= <l-expr> := <r-expr> ;
                   ::= <l-expr> ( [<r-expr>{ , <r-expr>}]  ) ;
                   ::= if <r-expr> then <stmtlist>
                       {  elsif <r-expr> then <stmtlist> }
                       [ else <stmtlist> ]
                       endif ;
                   ::= while <r-expr> loop
                           <stmtlist>
                       end ;
```

```
<l-expr>        ::= identifier

<r-expr>        ::= <l-expr>
                ::= <constant>
                ::= <l-expr> ( [<r-expr>{ , <r-expr>}] )
                ::= <r-expr> <binop> <r-expr>
                ::= <unop> <r-expr>
                ::= ( <r-expr> )

<constant>      ::= <integer-constant>
                ::= <real-constant>
                ::= <string-constant>
                ::= <boolean-constant>

<declarations>  ::= {<vardecl>}

<vardecl>       ::= {identifier , } identifier : <typedesc> ;

<typedesc>      ::= integer
                ::= real
                ::= boolean
                ::= string

<identifier>    ::= <letter>{<letter>|<digit>|_}(<letter>|<digit>)

<letter>        ::= A | B |...| Z | a | b ... | z

<digit>         ::= 0 | 1 |...| 9

<intconst>      ::= <digit>{<digit>}

<realconst>     ::= <digit>{<digit>} . {<digit>}
                ::= {<digit>} . <digit>{<digit>}

<booleanconst>  ::= true | false

<stringconst>   ::= " {(<letter>|<digit>|<printchar>)} "

<printchar>     ::= ' | ! | @ | # | $ | % | ^ | & | * | ( | ) | _ | -
                ::= + | = | [ | ] | { | } | ; | : | ' |, | . | < | >
```

```
                ::= ? | / | \ | | | ~

<binop>         ::= + | - | * | / | % | < | > | <= | >= | = | <>
                ::= and | or

<unop>          ::= - | not
```

For the translation from SBL to the UNIX shell programming language, the above description only specifies the syntactic mapping. The latter associates with each input (program from the source language), some structure which is in the domain of a second relation, the semantic mapping. This associates with the syntactic structure of each input, a string in some language (possibly the same language), which is considered to be the meaning of the original language. The semantics of the SBL have been described in plain English, in the section on the 'Structure of an SBL Program'. A more formal description can be given, but will not be included here.

8 Checkpointing and the Restart Mechanism

The existence of a restart mechanism, equivalent to the checkpointing mechanism in other programming environments, is one of the most important features for which the SBL and the SBL compiler have been designed. The atomicity of the SBL programs is at the level of the SB processing programs. If the execution of an SBL program was interrupted and has to be restarted, at most part of the SB program that was being executed when the interruption occured, will be repeated at restart. This is possible, as each SB processing program writes out data that is to be used (in general) by the next SB processing program.

The mechanism used to insure a correct restart at the point where the execution was interrupted, is based on saving the SBL program status, and the restart line number, before starting the execution of each SB processing program. The restart line number (RLN), is the line number in the SBL source program, of the built-in procedure call that corresponds to the SB processing program.

Given an SBL source program of the form:

```
<declarations>
<stmtlist>
```

the result of the compilation corresponds actually to a modified SBL source program, described here in pseudo-code (the pseudo-code parts are capitalized):

```
<declarations>
RLN : integer;
```

```
if (IT_IS_A_RESTART) then

        RESTORE PROGRAM STATUS AND THE RLN FROM THE LOG FILE

else

        RLN := 0

endif

<augmented_stmtlist>
```

The augmented statement list contains the original list of statements, in which the assignment statements, the repetitive, the conditional statements, and the built-in-procedure calls are modified as shown below, in a recursive definition.

Formally, this augmentation process can be represented as a set of productions:

```
<augmented-stmtlist>    ::=  <augmented-statement>{<augmented-statement>}

<augmented-statement>   ::=  null ;
                        ::=  <augmented-assignment-statement>
                        ::=  <augmented-if-stmt>
                        ::=  <augmented-while-stmt>
                        ::=  <augmented-procedure-call>
```

The augmented statements on the right-hand side are derived from the the non-augmented ones, as described further for each case. The representation of the 'augmented' productions exceeds the possibilities offered by the context-free grammars, as the augmented statements depend on attributes that are not dependent on the sets of terminals and non-terminals used in the language. In particular, the augmented statements depend on the position, represented by a line number, of the original statements in the SBL source program. Therefore, the augmented statements will be presented here in contrast with the respective original SBL statments.

An assignment statement, that was on line n_1 in the SBL source program (the line numbers will be dispalyed at the left, for clarity):

```
n1      <l-expr> := <r-expr>;
```

is translated as:

```
        if (RLN <= n1) then
```

```
                <l-expr> := <r-expr>;

        endif;
```

A repetitive statement:

```
n1      while (cond) loop

                <stmtlist>

n2      end;
```

becomes:

```
while(((cond) and RLN = 0) or (RLN >= n1 and RLN <= n2) loop

        <augmented_stmtlist>

end;
```

A conditional expression of the form:

```
n1      if (cond1) then

                <stmtlist>1

n2      elsif (cond2) then

                <stmtlist>2

n3      elsif (cond3) then

                <stmtlist>3

        ...

nk      else

                <stmtlist>k

        endif;
```

is translated as:

```
if (((cond1) and RLN = 0) or (RLN >= n1 and RLN < n2)) then
        <augmented-stmtlist>1
elsif (((cond2) and RLN = 0) or (RLN >= n2 and RLN < n3) then
        <augmented-stmtlist>2
elsif (((cond3) and RLN = 0) or (RLN >= n3 and RLN < n4)) then
        <augmented-stmtlist>3
...
else
        <augmented-stmtlist>k
endif;
```

Finally, a call to a built-in procedure is also augmented. A call to `print` is treated as if it were an assignment statement. For the other calls (to `envelope`, `fftinv`, `recip`, `fftexp`, `fftsynth`, or `rotation`), the initial call:

n_1 : `<l-expr> ( [<r-expr>{ , <r-expr>}] ) ;`

is treated as:

```
if (RLN <= n1) then
        SAVE PROGRAM STATUS AND RLN
        <l-expr> ( [<r-expr>{ , <r-expr>}] ) ;
        RLN := 0;
endif;
```

If the SBL program is executed from the beginning (not as a restart), the value of RLN is initialized to 0. This means that all the statements in the modified SBL program (with augmented statements), are executed exactly as in the original SBL program (with non-augmented statements).

If the SBL program is executed as a restart, the program status and the restart line number RLN are restored from the log file. These were saved right before starting the execution of the SB processing program whose completion was not successful.

In this case, all the statements in the SBL program are skipped up to the built-in procedure call for which the program status and the restart line number were saved last. Execution starts there, and when the SB program that is executed again completes, the restart line number, RLN, is assigned the value 0. This means that from here on, the execution of the SBL program with augmented statements, follows identically that of the original SBL source program.

It is clear that the restart mechanism works correctly, even in the cases where the restart point is inside the body of a repetitive statement, or in the body of a conditional statement.

As an example, consider the following SBL source program, with the line numbers displayed at the left. While this is not a real application SBL program, it illustrates most of the aspects related to the implementation of the restart mechanism in the SBL programs:

```
 1        /* Example for illustrating the restart mechanism       */
 2        i : integer;
 3        delta : real;
 4        b, c, d, e : boolean;
 5        enve, envi, envo : string;
 6
 7        null;
 8
 9        delta := 0.0005;
10        b := converged(0.01);
11        c := phaseext1("invcntlin", "reccntlin", "expcntlin", 1.0);
12        d := converged(delta);
13        e := phaseext1("invcntlin", "reccntlin", "expcntlin", 1.0);
14
15        if (((b and c) or (not b and not c)) and e) then
16                i := 2;
17        elsif (c) then
18                i := 3;
19        elsif (e) then
20                i := 4;
21        else
22                i := 5;
23        endif;
24
25        enve := "envexec";
26        envi := "envcntlin";
27        envo := "envcntlout";
28
29        i := 1;
```

```
30          while (i <= 10) loop
31
32                  i := i + 1;
33                  envelope(enve, envi, envo);
34
35          end;
36
```

The program compiled for the iPSC/860 (from which the parts that are not essential for illustrating the restart mechanism were removed) is:

```
#!/bin/csh -f

# Default initialization of the variables
  set delta = "0.0"
  set b = 0
  set c = 0
  set d = 0
  set e = 0
  @ i = 0
  set enve = ""
  set envi = ""
  set envo = ""

# Initialize the restart line number
  set _rln = 0

# Check if restart
  if ($#argv == 0) then

#   Restore the restart line number
    set _rln = `tail -1 $_logfile | grep 'PROGRAM_STATUS:' | awk '{print $2}'`
    if ($_rln == "") then
      echo "Cannot restore the restart line number from $_logfile . Bye"
      exit
    endif

    if ($_rln == 0) then
      echo "Restored  invalid restart line number 0 from $_logfile . Bye"
      exit
```

```
    endif

#   Restore the integer, boolean, and real variables
    set i = `tail -1 $_logfile | awk '{print $3}'`
    set e = `tail -1 $_logfile | awk '{print $4}'`
    set d = `tail -1 $_logfile | awk '{print $5}'`
    set c = `tail -1 $_logfile | awk '{print $6}'`
    set b = `tail -1 $_logfile | awk '{print $7}'`
    set delta = `tail -1 $_logfile | awk '{print $8}'`

    echo "RESTART"
    date
    echo "RESTART" >> $_logfile
    date >> $_logfile
    echo "PROGRAM STATUS : _rln  i e d c b delta"
    echo "PROGRAM_STATUS: $_rln $i $e $d $c $b $delta"

  else

    set _rln = 0

  endif

  if ($_rln <= 9) then
    set delta = "0.000500"
  endif

  if ($_rln <= 10) then
    set b = `~cryst/Prog/i860/Util/converge
      $_logdir/recip.new $_logdir/recip.old 0.010000`
  endif

  if ($_rln <= 11) then
    set c = `~cryst/Prog/i860/Util/phaseext1 1.000000 expcntlin reccntlin invcntlin`
#   Create recip.new
    echo "correlation coefficient for all the data 9.99" > $_logdir/recip.new

  endif

  if ($_rln <= 12) then
    set d = `~cryst/Prog/i860/Util/converge $_logdir/recip.new
```

```
      $_logdir/recip.old $delta`
  endif

  if ($_rln <= 13) then
    set e = `~cryst/Prog/i860/Util/phaseext1 1.000000 expcntlin reccntlin invcntlin`
#   Create recip.new
    echo "correlation coefficient for all the data 9.99" > $_logdir/recip.new

  endif


  if ((($b && $c) || ((!($b)) && (!($c)))) && $e)
    && $_rln == 0 || $_rln >= 15 && $_rln <= 16) then

    if ($_rln <= 16) then
      @ i = 2
    endif


  else if ($c && $_rln == 0 || $_rln >= 17 && $_rln <= 18) then

    if ($_rln <= 18) then
      @ i = 3
    endif


  else if ($e && $_rln == 0 || $_rln >= 19 && $_rln <= 20) then

    if ($_rln <= 20) then
      @ i = 4
    endif


  else

    if ($_rln <= 22) then
      @ i = 5
    endif


  endif
```

```
  set enve = "envexec"
  set envi = "envcntlin"
  set envo = "envcntlout"

  if ($_rln <= 29) then
    @ i = 1
  endif


  while (($i <= 10) && $_rln == 0 || $_rln >= 30 && $_rln <= 35)

    if ($_rln <= 32) then
      @ i = ($i + 1)
    endif

#   envelope ("envexec", "envcntlin", "envcntlout")

    if ($_rln <= 33) then

      set _exitline = `tail -1 $_logfile`
      set _exitstatus = `echo $_exitline | egrep 'End-of-' | wc -l`
      if ($_exitstatus == 0) then
        echo "$0 : error in the phase previous to envelope. Abort"
        exit
      endif

      echo "PROGRAM STATUS : _rln  i e d c b delta" >> $_logfile
      echo "PROGRAM_STATUS: 33 $i $e $d $c $b $delta" >> $_logfile
      newserver -c $_cubename >> envcntlout
      load -c $_cubename -H envexec
      startcube
      waitcube -c $_cubename < envcntlin

      set _rln = 0

    endif

  end
```

9 Auxiliary SB Procedures

The auxiliary SB procedures are meant to help the proper sequencing of the SB processing programs. Their correspondents in the SBL are currently the three built-in functions: `converged()`, `phaseext1()`, and `phaseext2()`. The names of the three auxiliary SB procedures are 'converge', 'phaseext1', and 'phaseext2'. Their locations in the 'cryst' directories (on all machine types), are:

```
~cryst/Prog/i860/Util/converge
~cryst/Prog/i860/Util/phaseext1
~cryst/Prog/i860/Util/phaseext2
```

The 'i860' may be replaced by 'Paragon', 'Sun', or 'RS6000' (the last too not currently available).

The interaction between the UNIX shell script that is the compiled SBL program, and these three procedures, is based on the possibility of assigning a value printed by a procedure (program) at the standard output, to a UNIX shell script variable, by:

```
set shell_variable = `program_name arguments`
```

The three auxiliary procedures perform each some actions, but finally they print either '1', or '0' at the standard output, to indicate success or failure, respectively. For this reason, from the SBL point of view, they are considered boolean functions, returning either `true`, or `false`.

The procedure **converge** takes one real argument on the command-line, that is the increment for which the overall correlation coefficients have to be tested after the execution of Recip, in order to determine if convergence has been reached in the iterative electron density averaging process, at a given resolution. The overall correlation coefficients for the last two runs of Recip are read by 'converge' from two files, 'recip.old' and 'recip.new', that are created by the UNIX shell script (the SBL compiled program). Before Recip is run for the first time, a file 'recip.new' is created, with an initial value of 9.99 for the overall correlation coefficient. It is important to know also that a call to `phaseext1()` in an SBL source program translates in the compiled code not only in the call to the 'phaseext1' SB program, but also in the creation of a new file 'recip.new', with a value of 9.99 for the overall correlation coefficient.

Then, at each run of Recip, 'recip.new' replaces 'recip.old', and 'recip.new' receives the latest value computed for the overall correlation coefficient. After reading the two correlation coefficient values, 'converge' subtracts them, and tests the absolute value of the difference, against its command line argument (for which acceptable user specified values are in the range from 0 to 0.5). If the difference is less than or equal to the command-line argument, it is considered that convergence has been reached, and the value '1' is printed at the standard output. If not, '0' is printed.

For example, if the desired increment to test for is 0.005, 'converge' will be invoked as:

```
~cryst/Prog/i860/Util/converge 0.005
```

The procedure **phaseext1** performs phase extension, after convergence has been reached at a given resolution, but maintaining the grid size and the mask file. A new value is computed for the upper resolution of the structure factors, and is written out to the control input files for the FFTinv, Recip, and FFTexp programs. New values are also computed for the structure factor indices, `hmax`, `kmax`, and `lmax`, and are written out to the control input file for the FFTinv program. The function takes four arguments on the command-line: a real argument that will be denoted by `incr`, which represents the increment used in computing the new values of `hmax`, `kmax`, and `lmax`, and three arguments of type string, representing, in order, the paths for the control input files of the FFTexp, Recip, and FFTinv programs. Assuming for example that `incr` is having the value 1.0, and that 'invcntlin', 'reccntlin', and 'expcntlin' are the names for the three control input files for FFTinv, Recip, and FFTexp respectively (it is assumed that the control input files are in the same directory with the UNIX shell script containing the call to 'phaseext1'), then 'phaseext1' is invoked as:

```
~cryst/Prog/i860/Util/phaseext1 incr expcntlin reccntlin invcntlin
```

The procedure prints '1' if phase extension is possible at the given grid size, and '0' otherwise. As a side effect, the control input files are also modified if phase extension is possible. At the SBL program level, just as for the `converged()` function, the value returned by `phaseext1()` has to be assigned to a boolean variable before being tested or used in a boolean expression of any kind (see the section on the 'Assignment Statement', in §2.9).

The logic implemented in the 'phaseext1' procedure is described further:

- Test the value of the `incr` command-line argument; acceptable values are in the range (0.0, 2.0].

- Read in the data in the control input files for FFTinv, Recip, and FFTexp.

- Read `rmax`, the high resolution, from the control input file for Recip.

- Read `a`, `b`, `c`, the unit-cell dimensions, from the control input file for Recip.

- Read `nx`, `ny`, `nz`, the number of grid steps in each direction, from the control input file for FFTinv.

- Set the real constant `k` to 3.0.

- Determine

```
maxdim = max(a, b, c)
indmax = maxdim / rmax
```

- Determine the real values:

```
hmax = a / rmax
kmax = b / rmax
lmax = c / rmax

hmaxnew =  hmax + incr
kmaxnew =  kmax + incr
lmaxnew =  lmax + incr

indmaxnew = indmax + incr
rmaxnew = maxdim / indmaxnew

ind = (maxdim * k) / rmaxnew
```

- Check if phase extension at the current grid size is possible:

```
if (min(nx, ny, nz) >= ind) then

       continue the phase extension process

else

       phase extension at the current grid size is not
       possible; print "0" at the standard output, and exit

endif
```

- Update the info for the control input file for FFTinv:

  - `rmaxnew - 0.05` replaces `rmax`
  - `hmaxnew`, `kmaxnew`, `lmaxnew`, are replacing the old values of the structure factor indices, `hmax`, `kmax`, and `lmax`

  Note: the slightly smaller value of the new `rmax` counterbalances the possible effects of the rounding errors when FFTinv computes the reflections that will be input to the Recip program; in this way, we avoid the situation in which certain reflections are not written out by the FFTinv program, but are expected by Recip.

- Update the info for the control input file for Recip:

  - `rmax` replaces `fcalcin`
  - `rmaxnew` replaces `rmax`

- Update the info for the control input file for FFTexp:

- `rmaxnew` replaces `rmax`
- Write out the control input files for FFTinv, Recip, and FFTexp

In case any error situation occurs, 'phaseext1' considers that the phase extension is not possible, prints '0' at the standard output, and exits.

The third auxiliary SB procedure, **phaseext2**, is provided to perform phase extension when the current grid size does not allow the use of 'phaseext1' anymore. This has to be combined with the generation of a new mask, too. The 'phaseext2' procedure is not operational as of Jan 12, 1994. From the SBL point of view, though, it is equivalent to the `phaseext1()` function, which will allow an easy implementation of the operational `phaseext2()`.

10 The SBL Compiler Implementation

The SBL compiler itself is a common tool, very similar to other, most often more complex such tools. The important aspect is that it translates a source program written in a simple, customized problem definition language, the Structural Biology Language, into a UNIX shell script capable to control the execution of a sequence of SB programs. Typically, a source program of 40 lines, translates into a UNIX shell script of more than 1000 lines, performing various tasks, one of the most important being the implementation of a restart mechanism. The actions performed by the compiled UNIX shell script are:

- Checks that the partition characteristics are acceptable for the target machine.
- Checks that the log file does not exist when a sequence of SB programs, specified in the form of an SBL program, is run for the first time.
- Checks that the log file does exist for a restart.
- Checks that all the files necessary for the execution of the SBL program exist, and that they have the right access permissions.
- Allocates the resources necessary for the run.
- If it is a restart, recovers all the partition characteristics,the restart line number, and the program status from the log file, and restarts execution at the right position in the SBL program.
- Before starting execution for each of the SB processing programs, saves the restart line number and the program status in the log file.
- For each SB processing program (Envelope, FFTinv, Recip, FFTexp, FFTsynth, or Rotation), or for the SB auxiliary programs ('converge', 'phaseext1', and 'phaseext2'), issues the appropriate commands for its execution; these depend on the target machine (iPSC/860, Paragon, Sun, or IBM RS6000), and on the OS.

- Records the values returned by the auxiliary SB procedures in shell variables, and uses them in controlling the sequencing of the SB processing programs, as specified in the SBL source program.

- Issues error messages or warnings whenever appropriate, and terminates the run, if necessary.

Through its functions, the shell script relieves the user (usually crystallographer or biologist) of the task of knowing the shell programming language details, and the specific commands on a variety of target machines. It does a variety of checkings that would require significant time to perform, creates the possibility of automatic cycling and phase extension, and, most important, implements a checkpoint and restart mechanism that allows completion of complex and time consuming jobs (consisting of several SB processing programs, in multiple nested loops), in several sessions.

The SBL compiler consists of three main parts: the lexical analyzer, the parser, that also handles the symbol table and builds a parse tree, and the code generator, with similar functions for various target machines. The error analysis is performed at each of these three levels.

Figure 1 illustrates the structure of the SBL compiler.

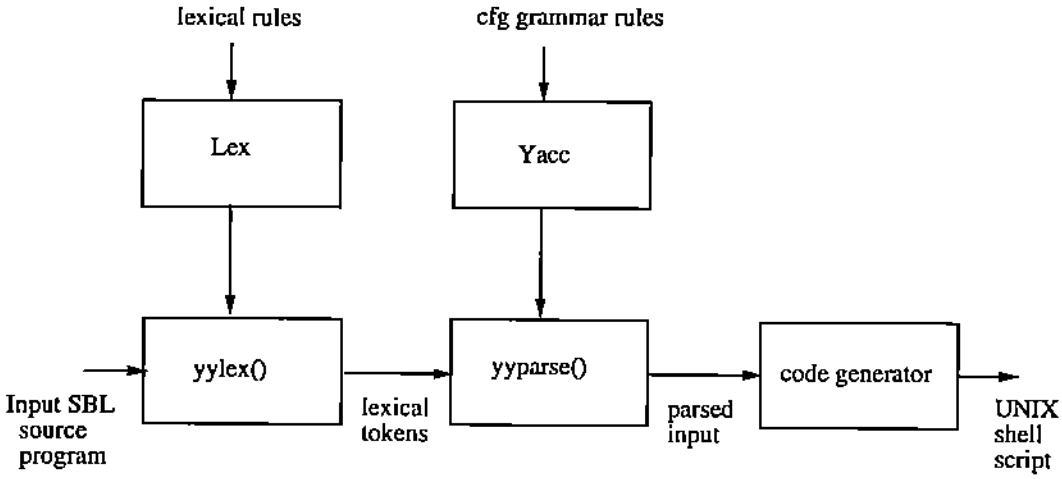


Figure 1. The structure of the SBL compiler

10.1 The Lexical Analyzer

The lexical analyzer was implemented using the Lex lexical analyzer-generator. The input to Lex is a table of regular expressions and corresponding program fragments. The deterministic finite automaton generated by Lex is segmenting the input in preparation for the parsing routine. The input to the lexical analyzer is a string of symbols from the alphabet of all the printable characters, plus the EOF character. The lexer, which is called by the parser, returns either a token, for certain combinations of characters that it recognizes in

the input, or the unknown character if it is not part of a known pattern, and issues an error message in this case.

Tokens are returned for keywords, identifiers, strings, integer constants, real constants, separators, parentheses, and operators, unary or binary.

10.2 The Syntactic Analyzer (The Parser)

The parser was produced using the Yacc tool, which accepts a broad class of specifications - the LALR(1) grammars, with disambiguating rules. Grammar rules derived directly from the context-free grammar describing the SBL, together with associated actions, were input to Yacc. The parser produced is a finite state machine, with a stack. It is capable of reading and remembering the next input token, called the 'lookahead' token. The current state is always the one on the top of the stack. Four actions are available to the state machine: shift (pop a state off the stack, and push another one on its top), reduce (replace the right-hand side of a grammar rule by its left-hand side), accept (indicates that the entire input has been seen, and that it matches the specification), and error (parsing can no longer be continued according to the specification). The actions associated with the grammar rules have on one side the role to detect errors that are related to the semantics of the language, and cannot be detected using only the cfg rules, and on the other side, to build the parse tree that will be used in code generation.

10.3 Data Structures

The lexical analyzer builds a simple symbol table, using a hashing technique, with the heads of the linked lists of structures of type NODE, in the array hashtab[] (figure 2).

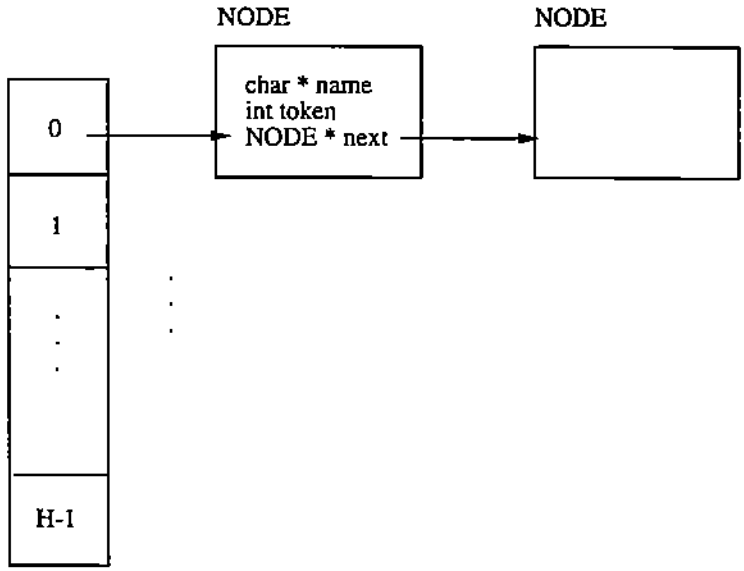


Figure 2. The symbol table built by the lexical analyzer.

The communication between the lexical analyzer and the parser is insured, other than through values returned by the lexical analyzer, also through values stored in the llval union, depending on the pattern identified by the lexer in the input:

```
union _llval
        { char * yystr; int yyint; float yyreal; NODE *yynodep; }
llval;
```

The tokens returned by the lexical analyzer to the parser are corresponding to keywords, separators, and SBL operators (listed in 'key_words.h' and 'tokens.h'). The keywords are stored in a table of structures, (containing each a token and a pointer to the actual name), used in binary search operations when checking whether an identifier is also a keyword of the SBL.

The parser rebuilds an enhanced symbol table, which contains besides the identifiers read from the input, also the built-in type names, and the names of the built-in procedures and functions. An entry in the parser symbol table is illustrated in figure 3.

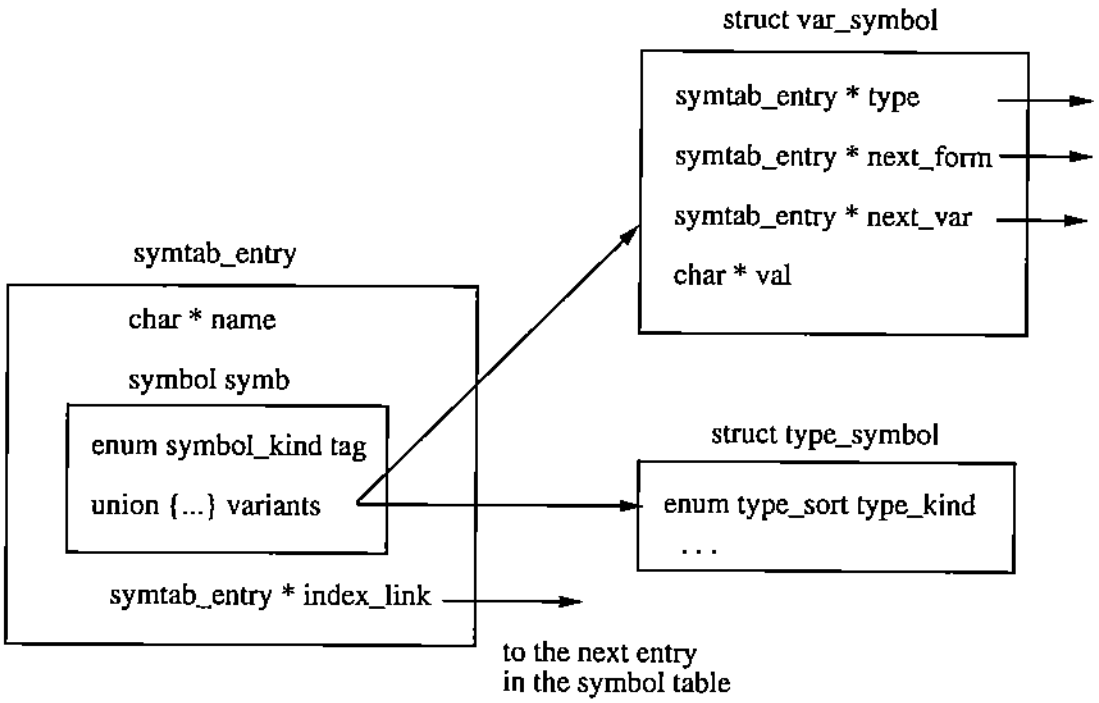


Figure 3. Entry in the symbol table for the syntactic analyzer.

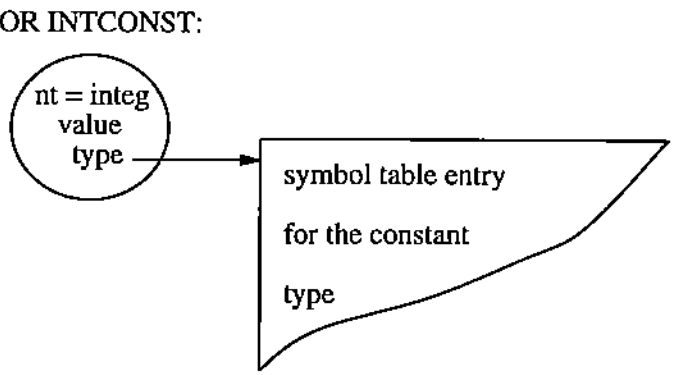
In figure 3, the 'variants' union is a pointer to either a structure for a type, or for a variable. The latter has also fields for the next formal if it is a formal parameter, or to the next variable, if in a variable declaration.

The syntactic analyzer builds a parse tree that will be used in code generation. Each node in the parse tree is materialized by a data structure having the following fields:

- the node type,
- a pointer to the symbol table entry for the type, if a variable or a built-in function,
- the value, if a constant of some type,
- a pointer to the string storing the name, if a variable,
- a pointer to the left child in the parse tree, and
- a pointer to the right child in the parse tree.

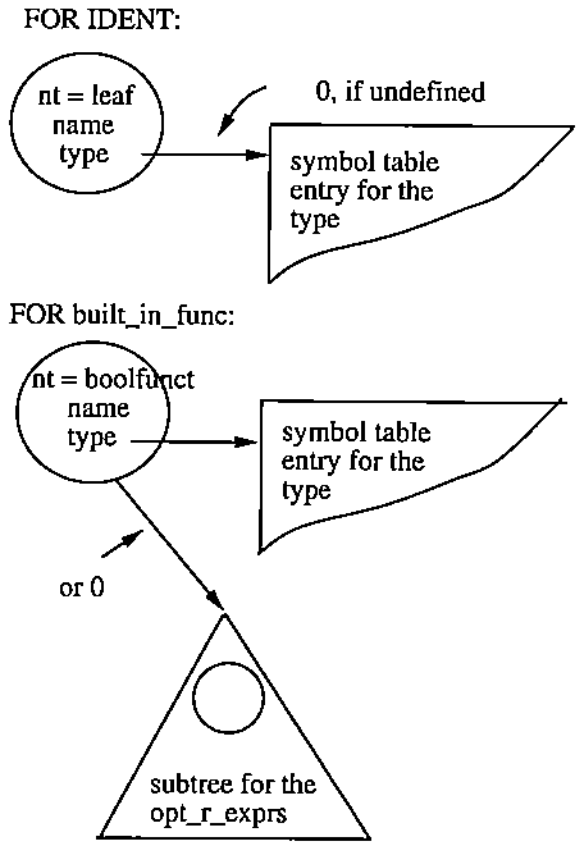
Figures 4, (a) to (i), illustrate parse tree fragments built when certain SBL constructs are recognized in the input by the syntactic analyzer. The `cfg` productions in their form accepted by the Yacc are given in full, but only some of the corresponding subtrees will be shown.

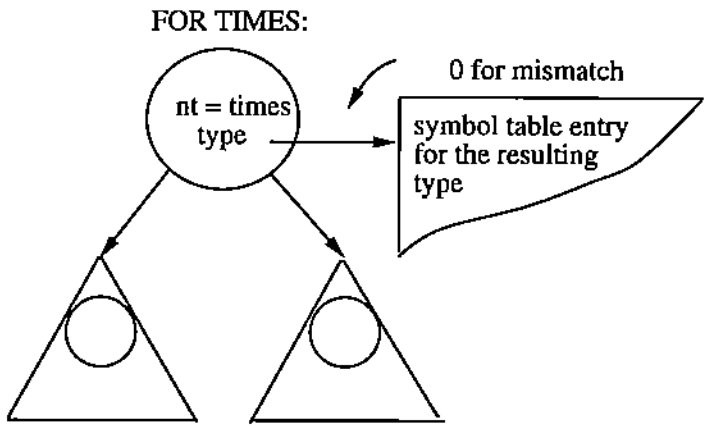
```
const: INTCONST
     | REALCONST
     | STRCONST
     | TRUE
     | FALSE;
```



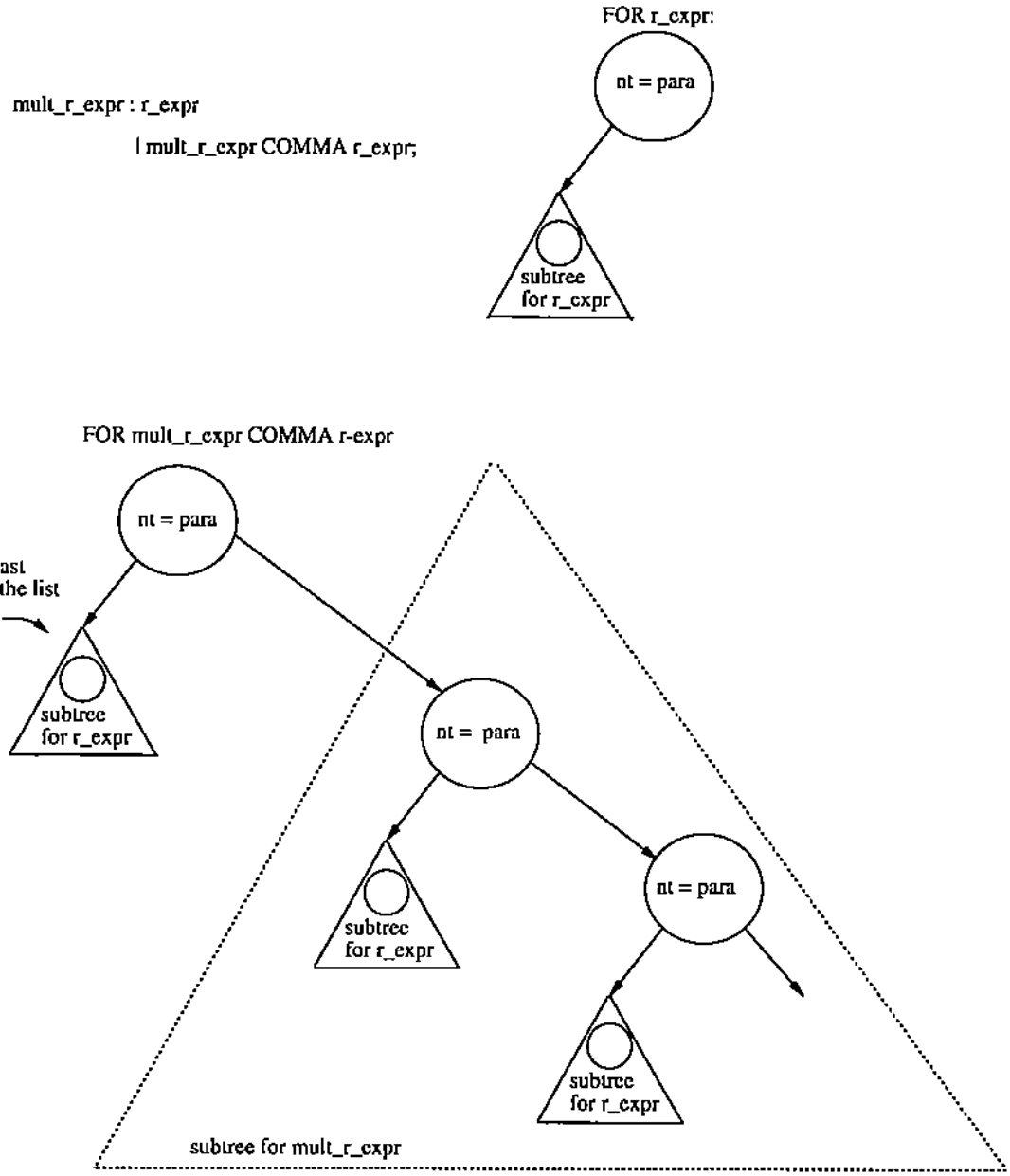
(a) Subtree built for a constant

```
r_expr: IDENT
      | const
      | built_in_func LPAREN opt_r_exprs RPAREN
      | MINUS r_expr
      | NOT r_expr
      | r_expr TIMES r_expr
      | r_expr DIV r_expr
      | r_expr MOD r_expr
      | r_expr PLUS r_expr
      | r_expr MINUS r_expr
      | r_expr LT r_expr
      | r_expr GT r_expr
      | r_expr LE r_expr
      | r_expr GE r_expr
      | r_expr EQ r_expr
      | r_expr NEQ r_expr
      | r_expr AND r_expr
      | r_expr OR r_expr
      | LPAREN r_expr RPAREN;
```



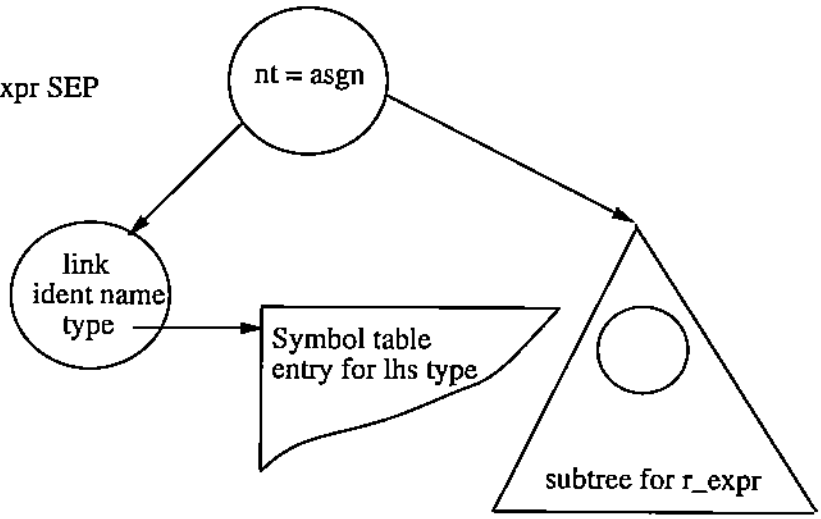


(b) Subtree built for an r_expression



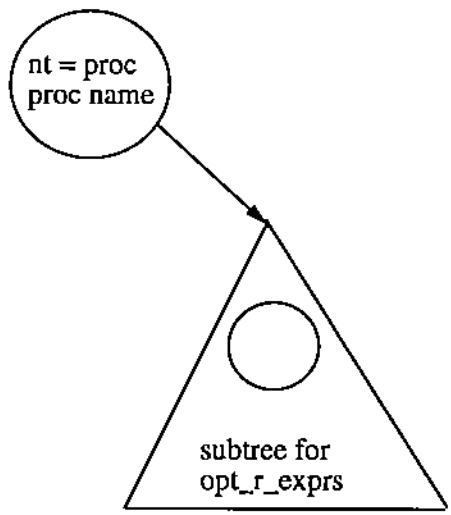
(c) Subtree built for multiple r_expressions, and for optional r_expressions

stmt: IDENT ASSIGN r_expr SEP



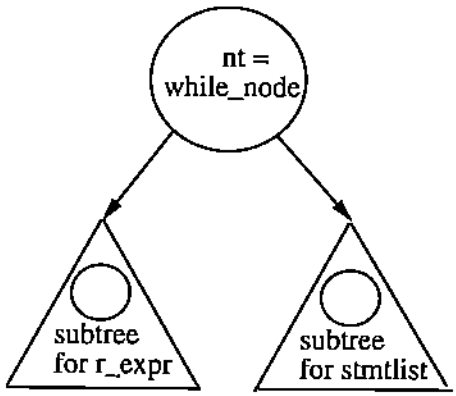
(d) Subtree built for the assignment statement

stmt: built_in_proc LPAREN opt_s_exprs RPAREN SEP



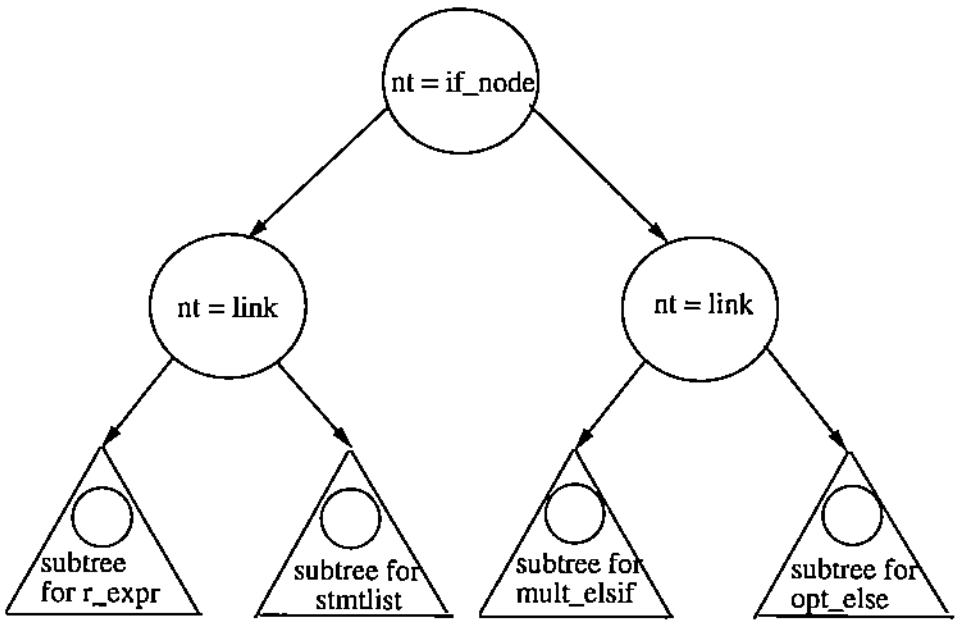
(e) Subtree built for the built-in procedure call

stmt: WHILE r_expr LOOP stmtlist END SEP



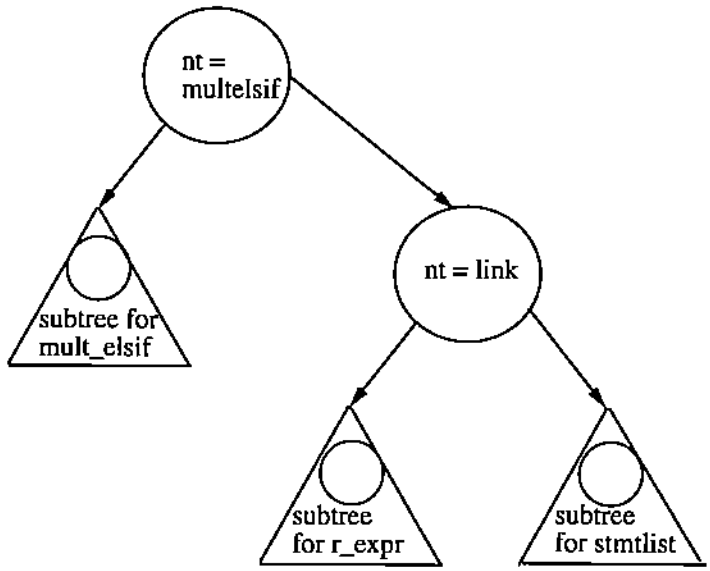
(f) Subtree built for the repeating statement

stmt: IF r_expr THEN stmtlist mult_elsif opt_else ENDIF SEP



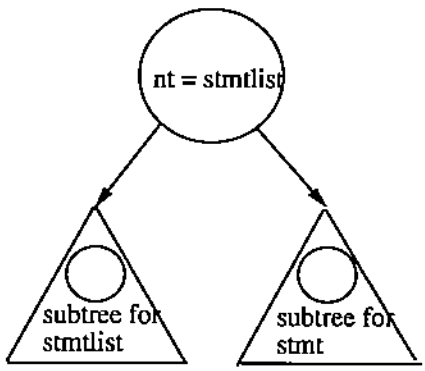
(g) Subtree built for the conditional statement

```
mult_elsif: /* empty, no subtree to build */
          | mult_elsif ELSIF r_expr THEN stmtlist
```



(h) Subtree built for multiple elsif part of the conditional statement

```
stmt_list: stmt /* copy subtree */
         | stmtlist stmt;
```



(i) Subtree built for a statement list

Figure 4. Subtrees in the parse tree, built for the different SBL constructs

10.4 The Code Generator

The code generator traverses the parse tree from the root, and generates the appropriate segments of code. As many checking operations are performed before the actual execution of the SBL program starts, a first traversal of the parse tree is dedicated to the generation of such initialization and checking operations. The next traversal is dedicated to the actual code generation.

Code is generated differently in some respects for different target machines. For example, the sequence of commands for the iPSC/860 machine:

```
getcube -c mycube -t16 > outfile
load -c mycube -H executable_path < infile
startcube -c mycube; waitcube
```

is replaced for the Paragon system, by:

```
executable_path < infile > outfile
```

The target machine is specified by a flag on the command line (i for iPSC/860, p for Paragon, s for Sun workstation, and r for IBM RS6000 workstation). The default target machine is the iPSC/860.

10.5 The SBL Compiler and the User Interface

The user interface invokes the compiler from a shell script, .sblpgen (SBL program generator), that is a virtually infinite edit - compile - view -rcp loop. The user has the option to exit after any step. If compilation of a program has terminated successfully, the compiled program can also be remotely copied (with 'rcp'), to the target machine.

The user interface packs together the SBL source program and the compiled UNIX shell script into a single file, when compilation ends. The source program appears as a comment at the beginning of the UNIX shell script. The exact structure of the script is:

- A first line containing the command '#!/bin/csh -f' that starts a new copy of the csh shell.
- A line specifying the number **n** of comment lines that follow, and correspond the the SBL source program.
- The **n** comment lines containing the source program.
- The compiled UNIX shell script.

When an existing program that has been created with the SB User interface is modified, it is unpacked first, and the user sees on the screen only the un-commented SBL source program.

Two functions, 'addsharp' and 'rmsharp' are used to pack and unpack the SBL programs ('#', or 'sharp', is marking comments in UNIX shell programs).

10.6 Functionality Extensions

The SBL compiler is functional as it is, and was tested on a significant number of real test programs. A number of extensions are possible, or even necessary.

1. Extending the code generation part for other target machines than the iPSC/860 and the Paragon.

2. Implementation of the `phaseext2()` function. The current procedure implementation uses the same number and types for the arguments to `phaseext2()`, as for `phaseext1()`. This makes it easy to change to a slightly different structure.

3. Extending the number of built-in procedures and/or built-in functions.

4. Adding new arguments to the built-in procedures, in order to specify the PEs to use when executing the corresponding SB programs. This will allow, without other supplementary language constructs, a certain degree of concurrency in executing different SB programs on different subpartitions of the same partition.

5. Adding an automatic consistency check mechanism: the SBL compiler can build the list of all the control input files in an SBL source program, and can then call an auxiliary SB procedure to check the consistency of all the control input data. This checking operation can be extended also to the headers of the input data files that are used in the execution of the SBL program.

11 SB User Interface Support for SBL Program Execution

The Structural Biology Language was defined to allow a simple and concise specification of an execution sequence of SB programs. The SB User Interface provides support for creating SBL program objects, and for creating, modifying, compiling, executing, and monitoring the execution of the SBL programs.

The SBL Program Menu

The 'SBL Program' menu is opened from the 'ITERATIVE ELECTRON DENSITY AVERAGING' menu.

Figure 5 illustrates these two menus, and also the 'SBL Program' object selection window.

For the 'SBL Program' menu, the 'SBL Program Selection' button opens the selection window seen in figure 5.

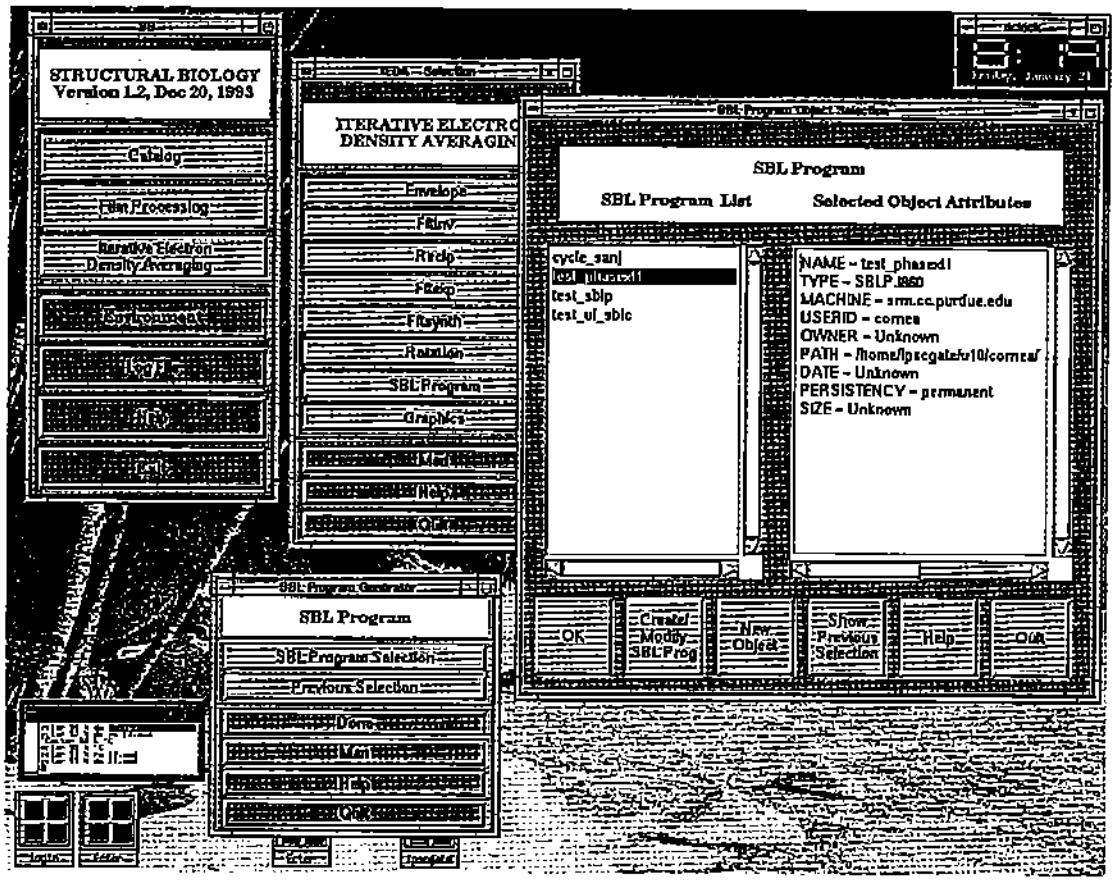


Figure 5. The main menu, the 'ITERATIVE ELECTRON DENSITY' menu, the 'SBL Program' menu, and the 'SBL Program' object selection window

The 'Previous Selection' button allows repeating the last successful selection from the current, or from a previous work session.

The 'Done' button opens an execution window, and the 'Man', 'Help', and 'Quit' buttons have the obvious roles. Pressing the 'Quit' button cancels any selection made during the current work session.

The SBL Program Object Selection Window

When opened, all the SBL program objects found in the user's object catalog, are displayed in the scrolled list window on the left. Selecting any of them with the mouse, will display its attributes in the scrolled text window on the right.

Once an object is selected, it can be 'okayed' by pressing the 'OK' button, which implies that it can be executed next.

The 'Create/Modify SBL Prog' button is used in creating, modifying, and compiling SBL programs. Its functions will be discussed in detail in the next section.

The 'New Object' button works as for the control input objects, allowing creation of an SBL program object, even if the underlying file does not exist yet, in the idea that it will be created as explained in the next section. This would not be possible starting from the 'Catalog' menu, just as it not possible for control input, or control output objects.

The 'Show Previous Selection' button displays the previous selections made in the current work session, in the scrolled text window on the right. This only clears from that window any object attributes, but does not cancel the current selection.

The 'Help' and the 'Quit' buttons have the same role as for the similar object selection windows.

Warning and wait messages are displayed for the 'SBL Program' menu and its descendents whenever appropriate. Figure 6 illustrates the wait message displayed when the SB User Interface remotely logs in to the machine hosting an SBL program, in order to check that it exists, and has the necessary access rights.

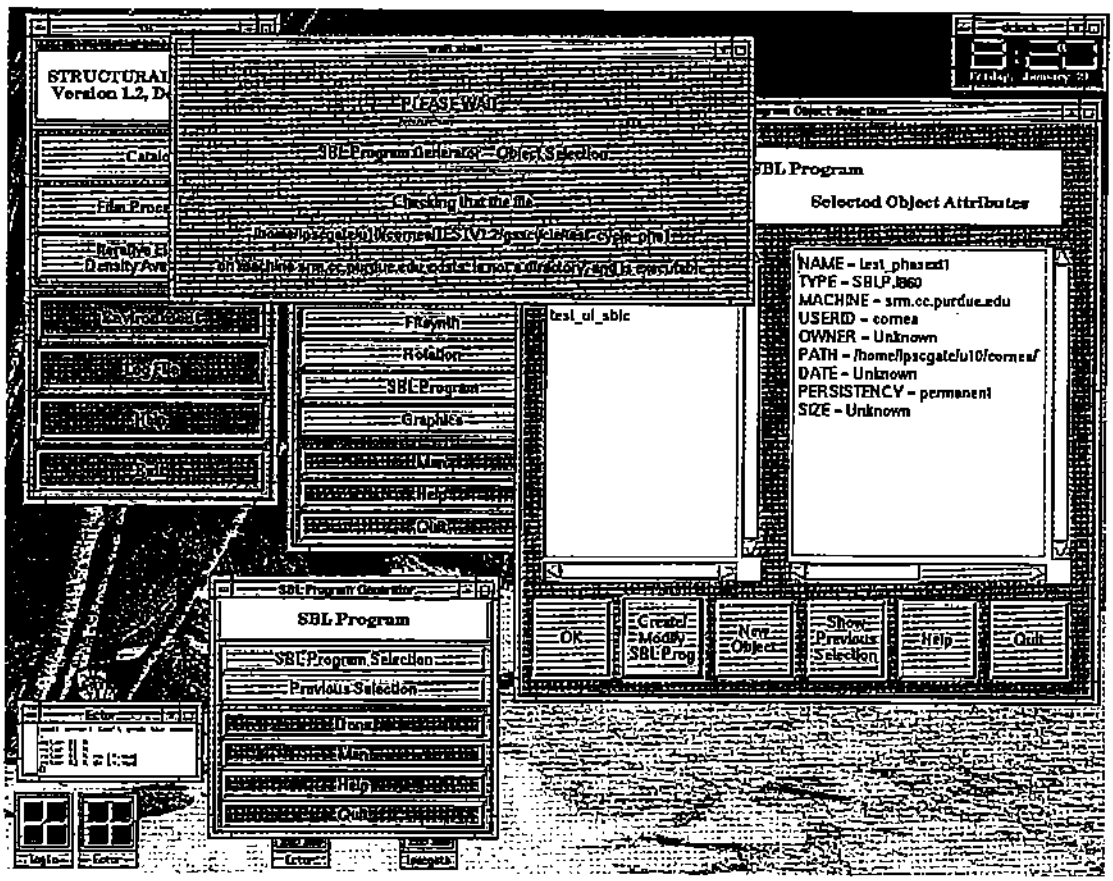


Figure 6. Wait message displayed while checking the access rights of an SBL program, on a remote machine

Creating, Modifying, and Compiling SBL Programs

New SBL programs can be created, or existing SBL programs can be modified, by selecting the corresponding object, and pressing the 'Create/Modify SBL Prog' button. In either case, an xterm window is opened, with a copy of the 'vi' editor started in it. If the SBL program does not exist, the 'vi' window is empty. If it exists, the SBL program is remotely copied from the machine that hosts it, is read into the editor buffer, and is displayed on the screen. Figure 7 illustrates the xterm window, with the 'vi' text editor started on an SBL source program.

When exiting from the text editor, the user is supposed to have typed in or modified an SBL source program.

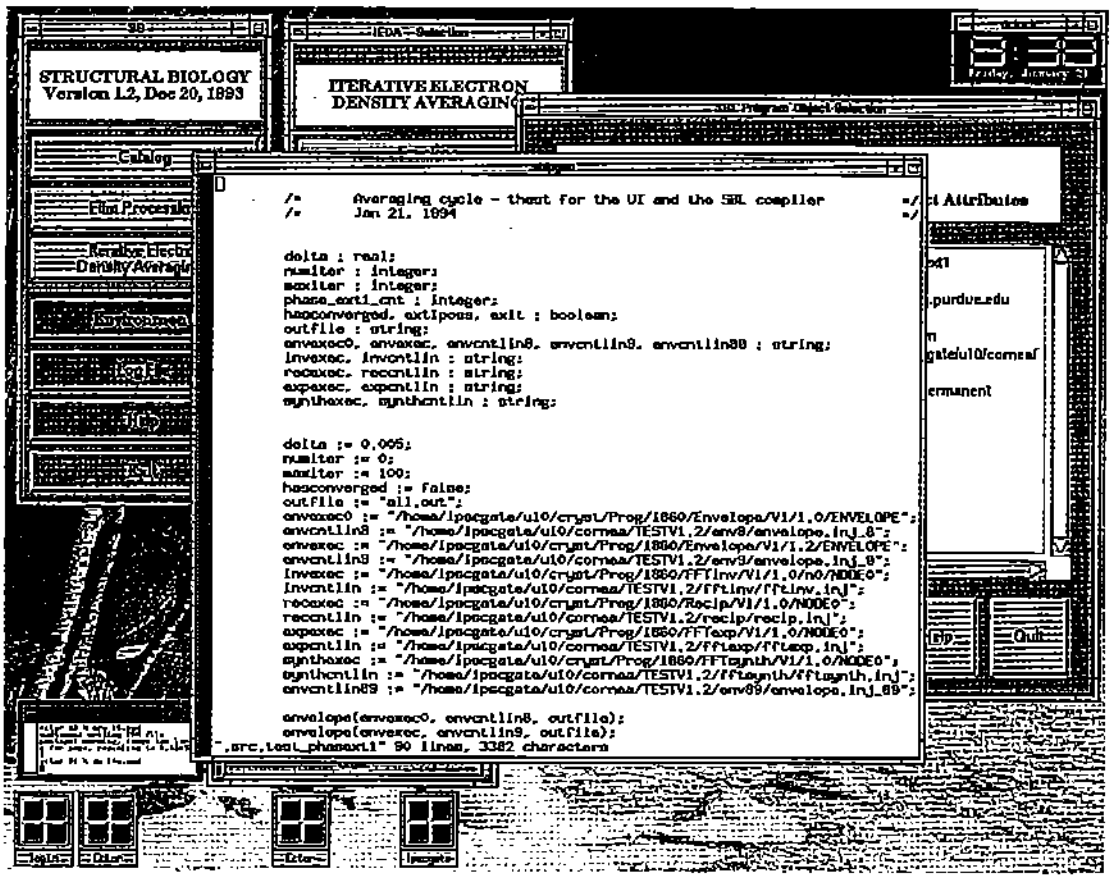


Figure 7. Xterm window, with 'vi' editor started on an SBL source program

Exiting automatically determines the compilation of the SBL source program. If there are errors, appropriate messages are displayed, and the user has the possibility to edit the file again, and recompile. If there were no errors, the compiled program can be remotely copied to the host machine, and/or viewed. Finally, the user has the option to exit from the xterm window, or to repeat the 'edit-compile-rcp-view' cycle.

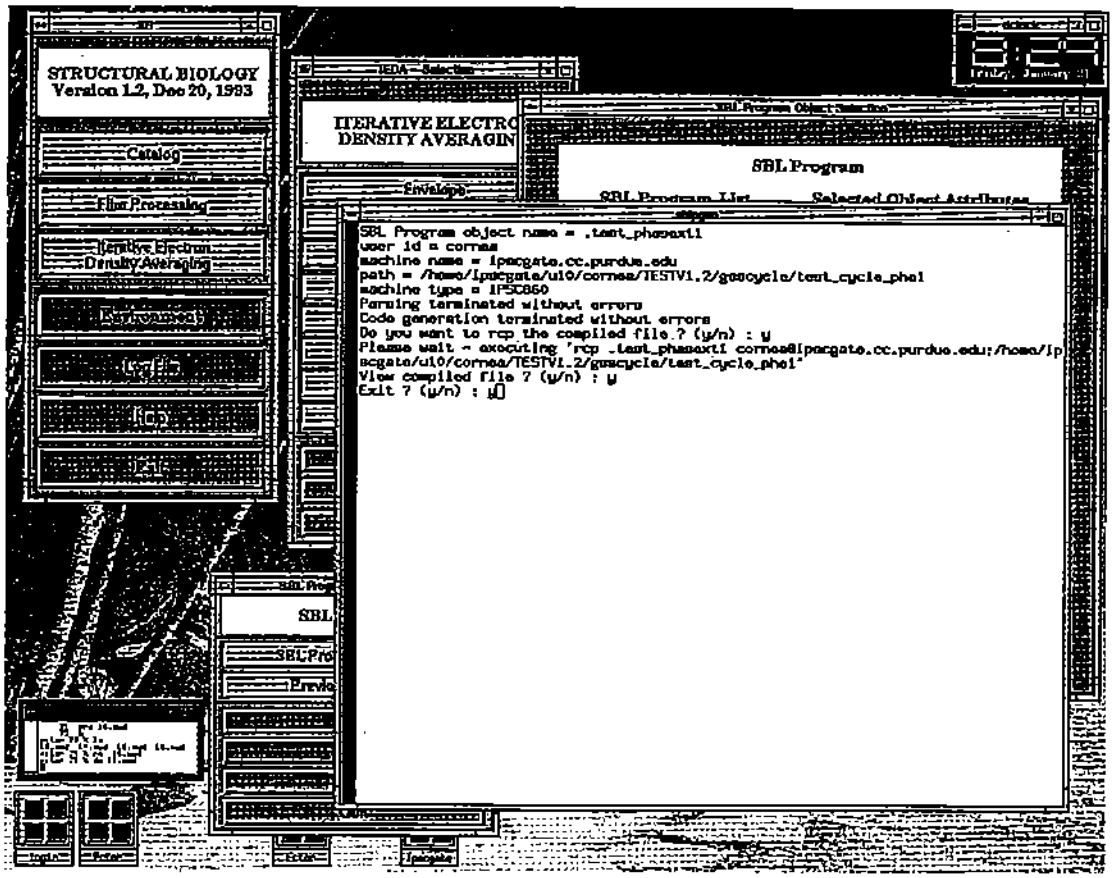


Figure 8. The xterm window after successful compilation of an SBL source program

Figure 8 shows an xterm window after successful compilation of an SBL source program. The result of the compilation is a UNIX shell program, that has the SBL source program inserted as a comment at its beginning. When an existing SBL program is modified, the entire UNIX shell program is copied locally, and the SBL source program is unpacked and displayed in the 'vi' editor window. The first part of the compiled UNIX shell program from the previous example is shown in figure 9.

Executing SBL Programs

SBL program execution can be initiated by specifying first a partition name, and the number of nodes to be used in the partition. This information can be filled out in an execution window, which also displays the SBL program's file pathname. Figure 10 presents also the SBL program execution window.

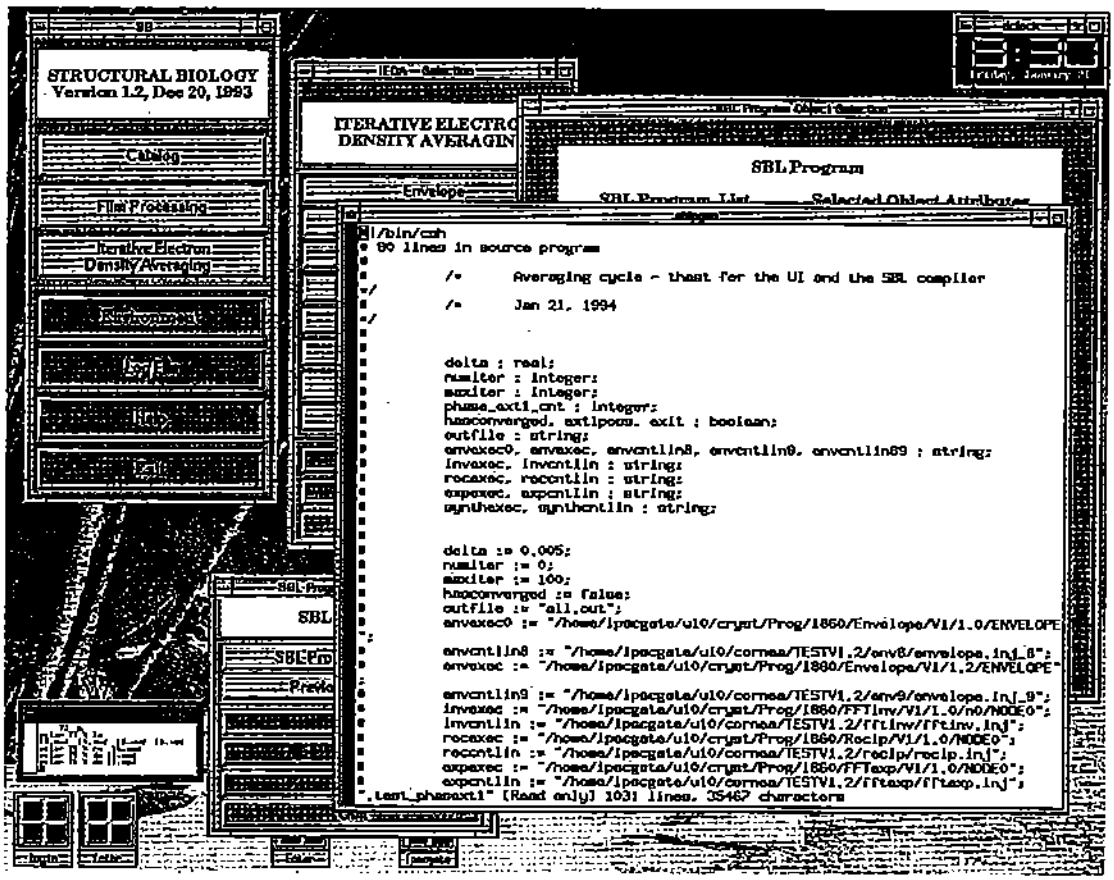


Figure 9. Compiled SBL program: the first part of the UNIX shell program contains as a comment, the SBL source program

The left-hand side column of the execution window contains push-buttons that display help messages.

The 'Run' button has to be used when an SBL program is executed from the beginning. If an SBL program's execution was interrupted for some reason, it can be restarted, using the 'Restart' button. In this case, the partition name and the number of nodes need not be specified, as they are restored by the UNIX shell program from a log file.

When using either 'Run' or 'Restart', an xterm window is opened, that displays all the messages going to the standard error output during the SBL program execution. The program execution is also started at this point.

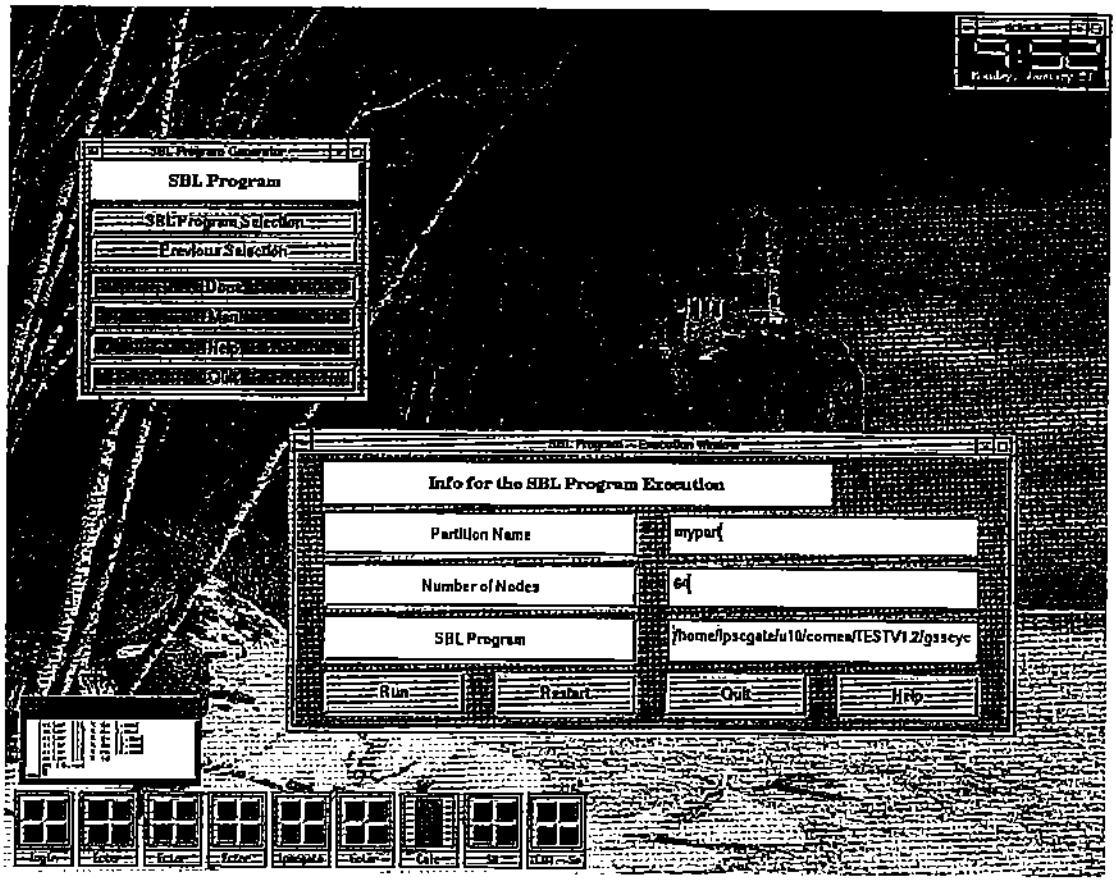


Figure 10. The SBL program execution window

The 'Quit' button will close the execution window, but not the xterm window, if it was opened.

Figure 11 shows the xterm window, after a failed SBL program execution: the 'Run' button tries to start the program from the beginning, but fails because the log file, named 'log.out', exists in the same directory with the UNIX shell program. Similarly, execution would fail as a restart, in the absence of the log file.

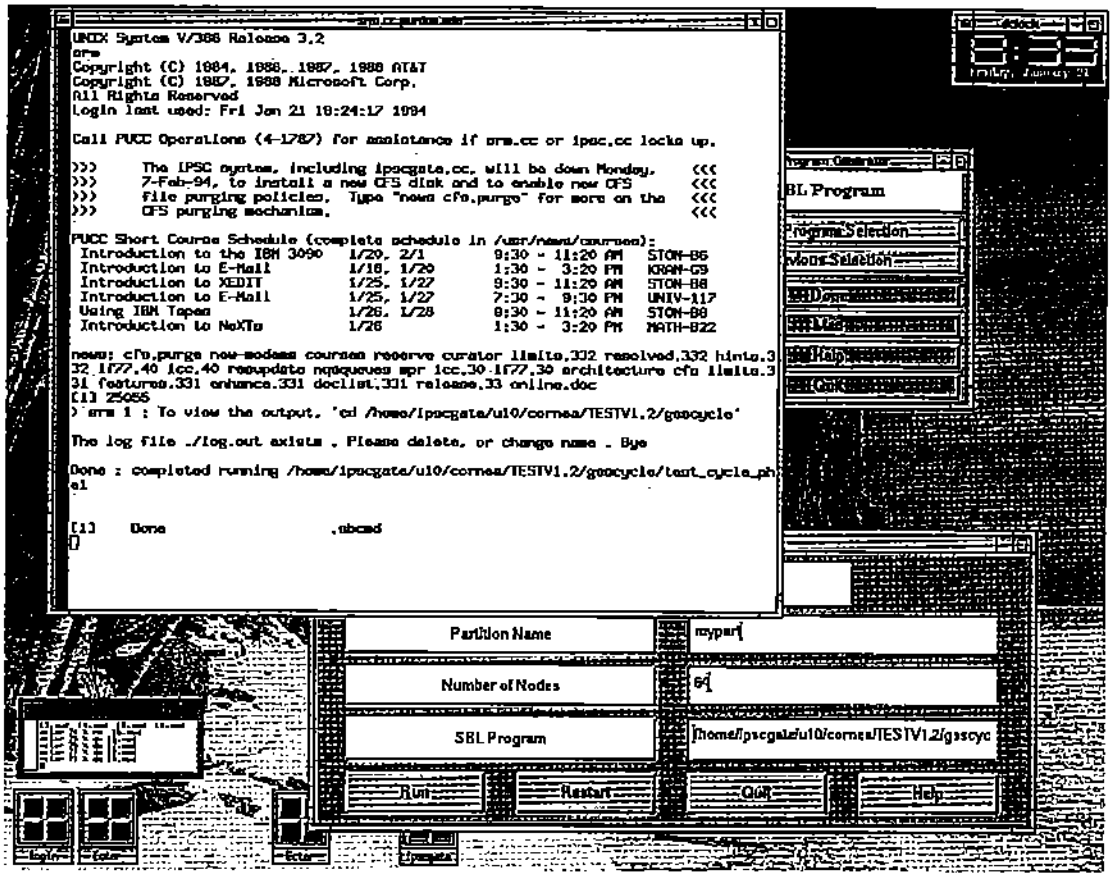


Figure 11. Xterm execution window for an SBL program

12 References

[**Aho 72**] A. V. Aho, J. D. Ullman, *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall, 1972

[**Bar 92**] N. Barkakati *UNIX Desktop Guide to X/Motif*, Hayden Books, 1992

[**Cha 92**] R. Chamberlain, *Paragon XP/S - An Applications Viewpoint*, Intel Technology Focus Conference, Timberline Lodge, Apr. 1992.

[**Fer 93**] P. M. Ferguson, *Motif Reference Manual for X Version 11 (Vol. Six B of 'The Definitive Guides to the X Window System')*, O'Reilly & Associates, Inc., 1993

[**Fis 88**] C. N. Fischer, R. J. LeBlanc, *Crafting a Compiler with C*, Benjamin/Cummings Publishing Co., Inc., 1988

[**Hea 91**] M.T. Heath, J.A. Etheridge, *ParaGraph : A Tool for Visualizing Performance of Parallel Programs*, Oak Ridge National Laboratory, Mathematical Sciences Section, Sep. 1991

[**Int 90-a**] Intel Corporation, *iPSC/2 and iPSC/860 User's Guide*, Jun. 1990

[**Mar 93**] D.C. Marinescu, J.R. Rice, M.A. Cornea-Hasegan, R.E. Lynch, M.G. Rossmann, *Macromolecular Electron Density Averaging on Distributed Memory MIMD systems*, Concurrency: Practice and Experience, Vol. 5(8), pp. 635-657, December 1993

[**Mar 94**] D. C. Marinescu, M. Cornea-Hasegan, C. Costian, I. Boier, *Towards Problem Solving Environment for High Performance Computing*, 1994 (under preparation)

[**Mor 93**] A. Morse, G. Reynolds, *Overcoming Current Growth Limits in UI Development*, C.A.C.M., Vol. 36, No. 4, April 1993

[**Nie 93**] J. Nielsen, *Noncommand User Interfaces*, C.A.C.M., Vol. 36, No. 4, April 1993

[**Nye 88-a**] A. Nye, editor, *Xlib Programming Manual for Version 11 (Vol. One of 'The Definitive Guides to the X Window System')*, O'Reilly & Associates, Inc., 1988

[**Nye 88-b**] A. Nye, editor, *Xlib Reference Manual for Version 11 (Vol. Two of 'The Definitive Guides to the X Window System')*, O'Reilly & Associates, Inc., 1988

[**Nye 90**] A. Nye, T. O'Reilly, *X Toolkit Intrinsics Programming Manual for OSF/Motif Release 1.2 (Vol. Four of 'The Definitive Guides to the X Window System')*, O'Reilly & Associates, Inc., 1990

[**Rei 88**] T. O'Reilly, V. Quercia, L. Lamb, *X Window System User's Guide for Version 11 (Vol. Three of 'The Definitive Guides to the X Window System')*, O'Reilly & Associates, Inc., 1988

[**Rei 90**] T. O'Reilly, editor, *X Toolkit Reference Manual for X Version 11 (Vol. Five of 'The Definitive Guides to the X Window System')*, O'Reilly & Associates, Inc., 1990

[**Ros 62**] M.G. Rossmann, D.M. Blow, *The Detection of Subunits Within the Crystallographic Asymmetric Unit*, Acta Crystallographica 15, 24, pp. 45 – 55, 1962

[**Ros 72**] M.G. Rossmann, editor, *The Molecular Replacement Method - A Collection of Papers on the Use of the Non-Crystallographic Symmetry*, Gordon and Breach Science Publishers, NY, London, Paris, 1972

[**Ros 90**] M.G. Rossmann, *The Molecular Replacement Method*, Acta Crystallographica A46, pp. 73 – 82, 1990

[**Ros 92**] M.G. Rossmann, R. McKenna, L. Tong, D. Xia, J. Dai, H. Wu, and H. Choi, *Molecular Replacement Real-Space Averaging*, Journal of Applied Crystallography, nr. 22, pp. 166 – 180, 1992.

[**Set 89**] R. Sethi, *Programming Languages - Concepts and Constructs*, Addison-Wesley, 1989

[**Sta 90**] R. Stansifer, *Lecture Notes in Programming Languages*, Purdue University, 1990

[**Tre 90**] P.C. Treleaven, *Parallel Computing Framework*, Parallel Computers - Object Oriented, Functional, Logic , edited by P.C. Treleaven, pp. 17 – 45, 1990

[**You 90**] D. A. Young, *The X Window System - Programming and Applications with Xt - OSF/Motif Edition*, Prentice-Hall, Inc., 1990