S A M P L E G E N E R A T O R

U S E R ' S M A N U A L

VERSION 2.0

S A M P L E G E N E R A T O R

U S E R ' S M A N U A L

VERSION 2.0

EMEC/ISA SAMPLE GENERATOR Software

Developed by: Rubén Hinojosa Chapel

User's manual written by Rubén Hinojosa Chapel

CONTENT

1. Introduction
2. Required Hardware
3. Installing Sample Generator
4. File Extensions
5. Beginning Sample Generator
6. The User Interface
7. Programming in SG

Appendices
A. Syntax of SG in BNF (Backus Normal Form)
B. Examples of programs in SG
C. Error Messages
D. Note frequencies and MIDI numbers

"Then I waited and waited and waited for the computation of the sound samples. The system for software synthesis was basically the same as now-a-days. Of course the control was much more primitive and it took much longer, a lot longer."

JOHN CHOWNING
(Music and Computers.
CD John Chowning)

1. INTRODUCTION

Sample Generator is a programming system based on IBM PC compatible Windows microcomputers, oriented to sound synthesis. It is a synthesizer by software. With the SG programming language, the user defines a **mathematic function** using algebraic expressions and classic control structures, like **for** and **while** loops and the **if** statement. So you can implement a wide range of synthesis algorithms. This function will be evaluated in an interval, normally thousands of mathematical points, for generating a 16 bit digital sound.

The result of the computation, i.e., the wave sound, can be used in several fields that need sounds, like multimedia applications and electroacoustic music composition, as the sound is available through a Sound Blaster or compatible sound card, or you can send the wave to a digital sampler (using another software, like Sample Vision or Sound Forge, not this one). The system can be used for teaching purposes and sound research too.

The wave formats used by Sample Generator are Microsoft Windows PCM wave file (.WAV) and Sample Vision wave file (.SMP), so you can use the wave files created by Sample Generator in other sound systems that make use of those formats, like IRIS's EDIT20/ARES MARS station software and most sound softwares for Windows.

The different typefaces displayed in this manual, refered to SG's programs, are for the following purposes:

Italics This typeface represents program comments.

Boldface SG's reserved words are set in this typeface.

`Regular` Any other kind of text.

In the Sample Generator's editor, all the texts are displayed in `Regular` typeface.

Sample Generator was developed at the EMEC (Estudio de Música Electroacústica por Computadoras, Studio for Computer Electroacoustic Music) by Rubén Hinojosa Chapel.

How to contact EMEC:

EMEC/ISA
Calle 120 # 1110
Playa, La Habana, Cuba
Phone: (537) 218975
Fax: (537) 336633
E-mail: isa@artsoft.cult.cu

2. REQUIRED HARDWARE

An IBM PC compatible with 486 DX2 66 MHz, 8 MB RAM, 1 MB free space in hard disk and a Sound Blaster card or compatible. You need also Microsoft Windows 3.11 or Windows 95. This configuration is enough for Sample Generator; a faster CPU is recommended though, mainly for complex calculations or computing long waves.

3. INSTALLING SAMPLE GENERATOR

To install Sample Generator onto your hard disk simply copy the SG folder from the installation diskette.

4. FILE EXTENSIONS

Sample Generator uses three different file name extensions:

♦**.SGP**: Use this for your SG source code files. You can use other file name extension, but by default .SGP is used. Its means: **S**ample **G**enerator **P**rogram.

♦**.WAV**: A Microsoft PCM wave file. Sample Generator uses this extension by default for saving and loading sound wave files.

♦**.SMP**: A Sample Vision wave file. Sample Generator can uses this extension for saving and loading sound wave files.

5. BEGINNING SAMPLE GENERATOR

CREATING YOUR FIRST PROGRAM

Execute Sample Generator and select the *File/New* option, so you will get an Edit window. You will be placed in the editor with the cursor in the upper left-hand corner. The following text will appear:

```
BEGIN
  Output :=
END.
```

Now you need to complete the source code. Remember you are defining a mathematic function to be evaluated again and again in every point of an interval. The following is the full text of your first program:

```
Program Test;
BEGIN
  Output := 25000 * Sin( 440 * Input );
END.
```

Place semicolons exactly where they appear in the manual, and make sure the last **end** is followed by a period. Don't worry about lowercase letters and uppercase letters: SG is not case sensitive. Use the Backspace key to make deletions, and use the arrow keys to move around in the Edit window.

ANALYZING YOUR FIRST PROGRAM

While you can type in and run this program without ever knowing how its works, we have provided a brief explanation here. The first line you entered gives the program name Test. This is an optional statement, but it is a good practice to include it.

The rest of the program contains the statements to be executed. The word **begin** signals the start of the program. The following statement contains the instruction to perform the calculations that generate the sound wave (`Output := 25000 * Sin( 440 * Input )`):

- **Output** is a system's variable that receive the result of the calculations.

- 25000 is the amplitude of the wave form. Since the system quantization is 16 bits, the maximum amplitude is 32767.

- **Sin** is the mathematic function Sin(x). Its generate a sinusoidal wave form.

- 440 is the sound frequency.

- **Input** is a system's variable that represent the time. Its value change automatically.

The program's execution starts with the first instruction after **begin**, only one in this case, and continues until **end.** is encountered.

SAVING YOUR FIRST PROGRAM

After entering your first program, it's a good idea to save it to disk. To do this, choose the *Save* command from the *File* menu or press Ctrl-S. By default your file will have been given the name NONAME1.SGP. You can rename it now by typing in TEST.SGP, or just TEST (the system will provide a SGP extension by default) and then pressing Enter. Any time you choose *File/Save* after that, your program will be saved as TEST.SGP. Programs in Sample Generator are saved in ASCII format (text files).

COMPILING AND RUNNING YOUR FIRST PROGRAM (RECORDING A WAVE)

Once you have typed and saved your first program, is time to compile it and execute it. Choose the *Record* command from the *Record* menu or press Ctrl-R. The system will calculate your first sound. A new window will be displayed: a Wave window. Its contains your new wave form.

PLAYING YOUR FIRST WAVE

To play the wave choose the *Wave/Play* command from the main menu, or press Ctrl-P. If you want to stop the sound, choose the Wave/Stop play command from the main menu, or press Ctrl-T.

SAVING YOUR FIRST WAVE

To save your wave choose the *Wave/Save as* command from the main menu and type in the file name (test.wav is recommended). You can save it in Microsoft PCM wave format (.WAV) or in Sample Vision wave format (.SMP).

6. THE USER INTERFACE

The user interface include the file windows, the wave windows, the toolbar and two types of menus: File Menu and Wave Menu.

In the File Window you can type in the source code of your programs. It works like most Windows's text editors.

The program typed in a File Window define a function for calculating a wave form. This wave form is painted in a Wave Window.

The File Menu include the following submenus: File, Edit, Record, Wave, Option, Window and Help. When you are in a File Window, the File Menu is activated.

The Wave Menu is like the File Menu, but without the submenus Edit and Record. It is activated when you are in a Wave Window. This difference between both menus prevent you from trying edit or record any program when you are not inside a File Window.

The structure of the general menu follow:

FILE

Use the File menu to open, save, close, and print new or existing SG program files.

New ..... Creates a new SG program file.
Open ..... Loads an existing SG program file.
Close ..... Closes the current SG program file window.
Save ..... Saves the current SG file using its current name.
Save As ..... Saves the current SG file using a new name.
Print ..... Sends the active file to the printer.
Print Setup .. Changes printer options and selects a printer from a list.
Exit ..... Closes all the windows and exits Sample Generator.

EDIT

Use commands from the Edit menu to manipulate text.

Cut ..... Removes a selected text and places it on the Clipboard.
Copy ..... Places a copy of the selected text on the Clipboard, leaving the original in place.
Paste ..... Copies the contents of the Clipboard into the File Editor Window.
Delete ..... Removes the selected text.
Select All ... Selects all the text in the File editor.

RECORD

Use commands from the Record menu to calculate the wave form or to test the function in a point.

Record ..... Evaluate the function in every point (sample size) for calculating the wave form.
Evaluate .... Displays a dialog box for evaluating the function in only one point, so you can test (debug) your function.

WAVE

Use commands from the Wave menu to manipulate Waves.

Open ..... Loads an existing mono 16 bit Microsoft wave file or Sample Vision wave file.
Save As ..... Saves the current wave file using a new name.
Play ..... Play the current wave file.
Stop Play ... Stop playing the current wave file.

OPTIONS

Use the Options menu to change system parameters.

Samples ..... Displays a dialog window for changing the sample size and/or the sampling rate, showing the length of the wave in seconds.

Sampling Rate .. Displays a dialog window for changing the sampling rate and/or the sample size, showing the length of the wave in seconds.

Hide Toolbar ... Hides or shows the Toolbar.
Set Font ..... Displays a dialog window for changing the Font parameters.

WINDOW

The Window menu contains commands for manipulating windows.

Tile ..... This option arranges the windows so they cover the entire desktop without overlapping one another.
Cascade ..... This option overlaps open windows so each is the same size as all others and only part of each underlying window is visible.
Arrange Icons .. Arrange all the icons of the minimized windows.
Close All ..... Closes all the opened windows.

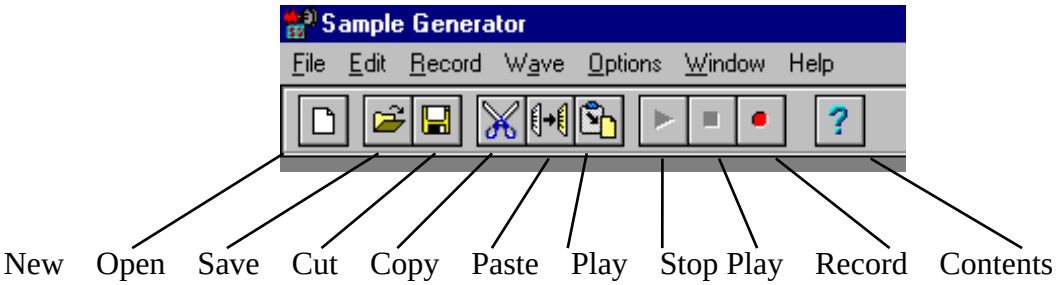
HELP

The Help menu provides access to on-line Help, which comes up in a special Help window.

Contents ..... Shows Sample Generator's on-line Help Contents screen. This screen summarizes the organization and contents of the Help system.
SG Language ..... Displays information about the SG language.
Note Frequencies . Shows the note frequencies of the tempered scale.
About ..... Displays information about the system and the author.

TOOLBAR

The Sample Generator Toolbar provides shortcuts for commands from the File, Edit, Record and Help menus.



7. PROGRAMMING IN SG

SG is a Pascal styled language: can be considered as a Pascal subset with some differences.

IDENTIFIERS

The names we give to variables, functions and programs are known as identifiers. The rules about identifiers are the following:

♦All identifiers must start with a letter or underscore (a..z, A..Z, or _). The rest of an identifier can consist of letters, underscores, and/or digits (0..9); no other character are allowed.

♦Identifiers are case-insensitive, which means that lowercase letters (a..z) are considered the same as uppercase letters (A..Z). For example, the identifiers freq, Freq and FREQ are identical.

♦Identifiers can be of any length, but only the first 32 characters are significant.

The following identifiers are reserved words:

```
or        and     not      program   init      begin
end       var     if       then      else      do
while     to      downto   for       until     repeat
exit      mod     sin      cos       exp       ln
sqrt      tan     cot      frac      int       pi
abs       rnd     pow      sgn       golden    square
triang    saw     input    output    incr      startvalue
```

DATA TYPES

SG has only one data type: the *real* type. You can not declare new types.

The following are real numbers:

43.271	-10.5	.5693
+0.345	30.00	-.1415

SG supports the usual set of binary arithmetic operator:

+	Addition
-	Subtraction
*****	Multiplication
/	Real division
Mod	Modulus (Rounded integer division)

Also, SG supports both unary minus (a + (-b)), which performs a two's complement evaluation, and unary plus (a + (+b)), which does nothing at all but is there for completeness.

The following are the predefined functions on the *real* type:

Sin(x) ..... Sine of the argument x.
Cos(x) ..... Cosine of the argument x.
Exp(x) ..... Exponential of the argument x.
Ln(x) ..... Natural logarithm of the argument x.
Sqrt(x) ..... Square root of the argument x.
Tan(x) ..... Tangent of the argument x.
Cot(x) ..... Cotangent of the argument x.
Frac(x) ..... Fractional part of the argument x.
Int(x) ..... Integer part of the argument x.
Abs(x) ..... Absolute value of the argument x.
Pow(x,y) ..... Power of x to the y.
Sgn(x) ..... Sign of x (-1, 0 or 1).
Square(x) ... Square wave.
Triang(x) .... Triangular wave.
Saw(x) ..... Sawtooth wave.

In the functions **sin**(x), **cos**(x), **tan**(x), **cot**(x), **square**(x), **triang**(x) and **saw**(x), x is a Real type expression and it is assumed to represent an angle in radians. The last three functions are, like the **sin**(x) function, periodics of period 2*Pi and maximum amplitude equal one (1). Is offered too the function without arguments **Rnd**. Its result is a Real random number within the range [0,1), i.e. the result is greater or equal 0 and less than 1. You can not define new functions as in others programming languages. May be in a future version of Sample Generator.

PREDEFINED REAL CONSTANTS

```
Pi = 3.1415...      (Pi)
Golden = (1 + Sqrt(5) ) / 2    (Golden section)
```

VARIABLES

A variable declaration embodies a list of identifiers that designate new variables. As the unique data type in SG is the **real** type, you don't need to give type explicitly for the variables.

An example of a variable declaration part follows:

```
VAR.
  X, X1, I, J, K;
  Frequency1;
  Frequency2;
```

Variables declared in this part are set to zero (0) at the beginning of the first execution of the program. The language offers predefined variables shared by the programmer (user) and the system. This variables are:

Input ..... Value used by the program (function) as argument.
Output ..... Value returned from the program (function).
StartValue .. **Input** is set to **StartValue** at the first execution.
Incr ..... **Input** is incremented by **Incr** in each running.

Although Sample Generator initializes the predefined variable **Incr** to the right value, you can use it for creating sounds changing its value at run-time and distortioning the time. A good example is the program Test8 in the appendix B.

EXPRESSIONS

Expressions are made up of operators and operands. An operand is a constant value, a value of a variable, the result of a function or the result of evaluating an operation. Most SG operators are binary, thats is, they take two operands; the rest are unary and take only one operand. Binary operators use the usual algebraic form, for example, a + b. A unary operator always precedes its operand, for example, -b. You can use the arithmetic operators we saw in the Data Types section or the following boolean operators and relational operators:

BOOLEAN OPERATORS

Not Negation (unary operator)
And Logical and
Or Logical or

This boolean operators, also called logical operators, work with the logical values True and False, allowing you to combine relational expressions and boolean expressions.

RELATIONAL OPERATORS

Relational operators allow you to compare two values, yielding a boolean result of True or False. Here are the relational operators in SG:

>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
=	Equal to
<>	Not equal to

Although the real type is the unique data type of the language, you can use boolean expressions, like:

```
(Input >= 3 * Pi) Or Not ( (Freq > 440) And (Freq < 540) )
```

Of course, you can not assign the result of a boolean expression to a variable.

PRECEDENCE OF OPERATORS

OPERATORS	PRECEDENCE	CATEGORIES
Not	first (high)	unary operator
*** / Mod And**	second	multiplying operators
+ - Or	third	adding operators
= <> > >= < <=	fourth (low)	relational operators

There are three basic rules of precedence:

- An operand between two operators of different precedence is bound to the operator with higher precedence.
- An operand between two equal operators is bound to the one on its left.
- Expressions within parentheses are evaluated prior to being treated as a single operand.

For the syntax of the expressions, look up the Appendix A.

INPUT / OUTPUT

In SG, input and output are made through the predefined variables **Input** and **Output**.

Input represent the argument of the function defined by the program. The following example show how to use it:

```
Var
A;
Begin
A := Input;
........
........
End.
```

The result of the calculation is returned through the predefined variable **Output**:

```
Var
  Freq;
Begin
  ........
  ........
  Output := Sin( Freq * Input );
End.
```

PROGRAM COMMENTS

Sometimes you want to insert notes into your program to remind you (or inform someone else) of what certain variables mean, what is the sampling rate or the sample size you want for the program, and so on. These notes are known as comments. SG, like most other programming languages, lets you put as many comments as you want into your program, without length limits.

A comment is any sequence of characters enclosed by curly braces, i.e., between the characters '**{**' and '**}**', and without the right curly brace (**}**) inside. An example is the following:

```
{ This is an example of a
  comment. See the examples
  of programs in Appendix B }
```

ASSIGNMENT STATEMENTS

Assignment statements replace the current value of a variable with a new value specified by an expression. The expression most be exclusively arithmetic. SG uses the symbol ':=' for the assignment. Some examples of assignment statement follow:

```
Freq := 440;
Freq := Freq + 0.01;
Output := 25000 * Sin( Freq * Input ) * Sin( 50 * Input )
```

COMPOUND STATEMENTS

The compound statement specifies that its component statement are to be executed in the same sequence as they are written. The compound statements are treated as one statement. **Begin** and **end** bracket the statements, which are separated by semicolons (;).

Here's an example of a compound statement:

```
Begin
  Temp := X;
  X := Y;
  Y := Temp;
End;
```

CONDITIONAL IF STATEMENTS

There are times when you want to execute some portion of your program when a given condition is True or not, or when a particular value of a given expression is reached. For this purpose SG offers the **If** statement. It can take the following generic format:

```
if expr then
  statement1
else
  statement2
```

where *expr* is any Boolean expression (resolving to True or False), and *statement1* and *statement2* are legal SG statements. If *expr* is True, then *statement1* is executed; otherwise, *statement2* is executed.

We most explain two important points about **if**/**then**/**else** statements:

First, **else** *statement2* is optional; in other words, this is a valid **if** statement:

```
if expr then
  statement1
```

In this case, *statement1* is executed if and only if *expr* is True. If *expr* is False, then *statement1* is skipped, and the program continues.

Second, what if you want to execute more than one statement if a particular expression is True or False? You would use a compound statement.

An example of the **if** statement follow:

```
if Input < 2 * Pi then
  Output := 20000 * Square( Freq * Input ) { statement1 }
else
 begin { This is a compound statement (statement2) }
   Freq := Freq + 0.01;
   Output := 25000 * Square( Freq * Input );
 end;
```

Note that *statement1* has no semicolon (;) at the end. See the programs RingPhone and Test2 in the appendix B for a good example of how to use both types of the **If** statement.

LOOPS

Just as there are statement (or groups of statements) that you want to execute conditionally, there are other statements that you may want to execute repeatedly. This kind of construct is known as a loop. There are three basic kinds of loop: the **while** loop, the **repeat** loop, and the **for** loop. We'll cover them in that order. If the number of repetition is known beforehand, the **for** statement is the appropiate construct. Otherwise, the **while** or **repeat** statement should be used.

WHILE STATEMENTS

You can use the while loop to test for something at the beginning of your loop. The format of the while statement is:

```
while expr do statement
```

where *expr* is a Boolean expression and *statement* is either a single or a compound statement. The **while** loop evaluates *expr*. If it's True, then *statement* is executed, and *expr* is evaluated again. If *expr* is False, the **while** loop is finished and the program continues.

An example of how to use the **while** loop is the program TestWhile, in the appendix B.

REPEAT STATEMENTS

The second loop is the **repeat.until** loop. Here's the generic format for that statement:

```
repeat
  statement;
  statement;
  ....
  statement
until expr
```

In the **repeat.until** loop, everything between **repeat** and **until** is repeated until the expression following **until** is True. There are three major differences between the **while** loop and the **repeat** loop. First, the statements in the **repeat** loop always execute at least once, because the test on *expr* is not made until after the repeat occurs. By contrast, the **while** loop will skip over its body if the expression is initially False.

Next the **repeat** loop executes *until* the expression is True, where the **while** loop executes *while* the expression is True.

Finally, the **repeat** loop can hold multiple statements without using a compound statement. You don't have to use **begin..end** in the **repeat** loop, but in the while loop you will need the **begin..end** to hold multiple statements.

Again, be careful to note that the **repeat** loop will always execute at least once. A **while** loop may never execute depending on the expression.

An example of how to use the **repeat** loop is the program TestRepeat, in the appendix B. Notice the differences between the TestWhile and the TestRepeat programs.

FOR STATEMENTS

The **for** loop is the one found in most major programming languages. Basically, you execute a set of statements some fixed number of times while a variable (known as the *index variable*) step through a range of values. Here's the generic format of the for loop statement:

```
for index := expr1 to expr2 do statement
```

where *index* is a variable, *expr1* and *expr2* are expressions, and *statement* is a single or compound statement. *Index* is incremented by one after each time through the loop. You can also decrement the index variable instead of incrementing it by replacing the reserved word **to** with the reserved word **downto**:

```
for index := expr1 downto expr2 do statement
```

The **for** loop is equivalent to the following code:

```
index := expr1;
while index <= expr do
begin
  statement;
  ...
  index := index + 1;
end;
```

The main drawback of the **for** loop is that it only allows you to increment or decrement by one. Its main advantages are conciseness and a faster looping.

An example of how to use the **for** loop is the program TestFor, in the appendix B. It is a version of the program TestWhile, but with the **for** loop. Notice the faster calculation of the wave form. Another examples are the programs Test9 and Test10 (additive synthesis).

EXIT STATEMENTS

Sometimes, after a calculation inside a loop or the **if** statement or any other place, you need to terminate and leave the program. The **exit** statement may be used to abort the execution of the program. It's generic format is very simple. Just type the reserved word **exit**:

```
begin
  statement;
  statement;
  exit;
  statement;
end.
```

PROGRAMS

In the SG language the structure of a program is a *header* followed by a *block*. The *header* can be written or omitted, and consist of the reserved word **program** followed by an identifier.

Example:

```
program MyFirstProgram;
```

The structure of the *block* is:

```
<variables declaration part>
<initialization part>
<body of the block>
```

The *variables declaration part* was analyzed in the variables section, and can be written or omitted.

The generic format of the *inicialization part* is the reserved word **init** followed by statements, normally assignment statements:

```
init
  statement;
  statement;
```

and can be written or omitted. Its objective is letting the program to initialize its variables. This part is executed only in the first running of the program.

The *body of the block* most be written always. Its generic format follow:

```
begin
  statement;
  statement
  ...
  statement;
end.
```

Notice the period (.) after the **end**.

In the *body of the block* most be written at least one time the following sequence:

```
Output := expr
```

because this is the way of the program to return the result of the calculation to the Sample Generator system. *Expr* is an arithmetic expression. An example follow:

```
Begin
  Output := 10000 * Sin( 110 * Input )
End.
```

Programs in SG have the file extension SGP, which means Sample Generator Program. You can see some examples of SG's programs in the appendix B.

A. Syntax of SG in BNF (Backus Normal Form)

```
<program> ::= <prog_header> <block>

<prog_header> ::= program identif ';' |

<block> ::= <var_decl_part> <init_part> <body_block>

<init_part> ::= <init_word> <inst_list> |

<init_word> ::= init

<body_block> ::= <block_start> <inst_list> end

<block_start> ::= begin

<comp_inst> ::= begin <inst_list> end

<inst_list> ::= <inst_list> ';' <inst> | <inst>

<gral_inst> ::= <comp_inst> | <inst>

<inst> ::= <repeat_loop>|
           <asignment>  |
           <for_loop>   |
           <while_loop> |
           <cond_inst>  |
           <exit_inst>  |
            ';'         |

<var_decl_part> ::= var <var_decl_list> ';' |

<var_decl_list> ::= <var _decl> |
                    <var_decl_list> ';' <var _decl>

<var _decl> ::= <var_list>

<var_list> ::= identif |
               <var_list> ',' identif

<cond_inst> ::= <then_part> <gral_inst> |
                <if_clausule> <gral_inst>
```

```
<then_part> ::= <if_clausule> <gral_inst> else

<if_clausule> ::= if <expr> then

<while_loop> ::= <condition> <gral_inst>

<condition> ::= <while_word> <expr> do

<while_word> ::= while

<for_loop> ::= <loop_ctrl_to> do <gral_inst> |
               <loop_ctrl_downto> do <gral_inst>

<loop_ctrl_to> ::= <loop_start> to <expr>

<loop_ctrl_downto> ::= <loop_start> downto <expr>

<loop_start> ::= for <var> ':=' <expr>

<repeat_loop> ::= <repeat_word> <inst_list> until <expr>

<repeat_word> ::= repeat

<asignment> ::= <var> ':=' <expr>

<exit_inst> ::= exit

<expr> ::= <simple_exp> |
           <simple_exp> <rel_op> <simple_exp>

<simple_exp> ::= <simple_exp> '+' <term> |
                 <simple_exp> '-' <term> |
                 <simple_exp> or <term>  |
                 <term>     |
                 '+' <term> |
                 '-' <term> (uminus)

<term> ::= <term> and <factor> |
           <term> '/' <factor> |
           <term> mod <factor> |
           <term> '*' <factor> |
           <factor>

<factor> ::= not <factor>      |
             <var>             |
             <unsigned_const>  |
             (' <expr> ')'     |
             <sin_desig>       |
             <cos_desig>       |
             <exp_desig>       |
             <ln_desig>        |
             <sqrt_desig>      |
             <tan_desig>       |
```

```
                <cot_desig>      |
                <frac_desig>     |
                <int_desig>      |
                <pi_desig>       |
                <abs_desig>      |
                <rnd_desig>      |
                <pow_desig>      |
                <sgn_desig>      |
                <golden_desig>   |
                <square_desig>   |
                <triang_desig>   |
                 <saw_desig>

<rel_op> ::= '=' |
             '<>'|
             '<' |
             '>' |
             '<='|
             '>='

<var> ::= identif

<unsigned_const> ::= realconst

<sin_desig> ::= sin '(' <expr> ')'

<cos_desig> ::= cos '(' <expr> ')'

<exp_desig> ::= exp '(' <expr> ')'

<ln_desig>  ::= ln '(' <expr> ')'

<sqrt_desig> ::= sqrt '(' <expr> ')'

<tan_desig> ::= tan '(' <expr> ')'

<cot_desig> ::= cot '(' <expr> ')'

<frac_desig> ::= frac '(' <expr> ')'

<int_desig> ::= int '(' <expr> ')'

<pi_desig>  ::= pi

<abs_desig> ::= abs '(' <expr> ')'

<rnd_desig> ::= rnd

<pow_desig> ::= pow '(' <expr> ',' <expr> ')'

<sgn_desig> ::= sgn '(' <expr> ')'

<golden_desig> ::= golden
```

```
<square_desig> ::= square '(' <expr> ')'

<triang_desig> ::= triang '(' <expr> ')'

<saw_desig> ::= saw '(' <expr> ')'

<digit> ::= '0'..'9'

<letter> ::= 'a'..'z' |
             'A'..'Z' |
             '_'

<identif> ::= letter(letter | digit)*

<realconst> ::= (digit)+ |
                (digit)* '.' (digit)+
```

B. Examples of programs in SG.

```
program test;
begin
  output := 25000 * sin( 440 * input )
end.

------------------------------------------------------------

program test1; { 30000 samples }
init
  incr := 0.06;
  startvalue := 0;

begin
  output := 5000 * sin( input * sin( input / 60 ) ) *
                   ( 4 + sin( input / 70 ) );
  incr := incr + 0.0001;
end.

------------------------------------------------------------

program test2;
{ samples = 12288, sampling rate = 22050 }
var
   fo;
init
   fo := 27.5;
   incr := pi / 1024;
begin
   if input < 2 * pi then
     output := 20000 * sin( fo * input )
   else if input < 4 * pi then
     output := 21000 * sin( 2 * fo * input )
   else if input < 6 * pi then
     output := 22000 * sin( 4 * fo * input )
   else if input < 8 * pi then
     output := 23000 * sin( 8 * fo * input )
   else if input < 10 * pi then
```

```
    output := 24000 * sin( 16 * fo * input )
  else
    output := 25000 * sin( 32 * fo * input ) ;
end.

-------------------------------------------------------

program test3; { samples = 15000 }
init
  startvalue := 0;
  incr := 0.1;
begin
  if input < 1800 then
    output := 15000 * sin( 10 * sin( input / 300 ) * input )
  else
    output := 20000 * sin( 20 * sin( input / 200 ) * input );
end.

-------------------------------------------------------

program test4; { power funtion as envelope }
{ samples = 22050 }
begin
  output := 40 * pow( input, 3.3 ) * sin( 440 * input );
end.

-------------------------------------------------------

program test5;      { exponential function as envelope }
{ samples = 22050 }
begin
  output := 20 * exp( input ) * sin( 440 * input );
end.

-------------------------------------------------------

program test6; { synthesis by phase distortion }
var
  phase;
init
  phase := 0;
begin
  phase := phase + 4 * sin( 5 * input );
  output := 20000 * sin( phase + 55 * input );
end.

-------------------------------------------------------

program test7; { random sound }
begin
  output := 25000 * rnd * sin( 220 * input + rnd );
end.

-------------------------------------------------------

program test8; { modifying the time increment }
init
  incr := 0.007;
```

```
begin
  incr := incr + 0.00001;
  output := 10000 * square( input + 10 * sin( input / 100 ) ) *
                  ( 1 + sin( input / 140 ) + sin( input / 80 ) );
end.

------------------------------------------------------------

program test9; { additive synthesis ( organ ) }
var
  i;

begin
  output := 0;
  for i := 1 to 5 do
    output := output + sin( 55 * pow(2,i) * input );
  output := output * 1500 * (4 + sin( 10 * input ) );
end.

------------------------------------------------------------

program test10; { additive and random synthesis }
var
  i, s;

init
  startvalue := 0;
  s := 0;
begin
  output := 0;
  s := s +  .001;
  for i := 1 to 5 do
    output := output + sin( i * ( 440 + rnd/20 - s ) * input );
  output := 7000 * output;
end.

------------------------------------------------------------

program test12;  { synthesis by fm }
                 { sampling rate = 22050 }
 var
   a, b;
   frec;
 init
   incr := 2 * pi / 22050;
   a := 0;
   b := 0;
   frec := 220;
 begin
   output := 15000 * sin( a + b * sin( frec * input )  );
   if input < 2.42 then
     a := a + pi / 1000
   else
     a := a - pi / 2000;
   b := 2 * a;
   frec := frec + 0.005;
 end.

------------------------------------------------------------
```

```
program test13;
{  samples = 15000 }
init
   startvalue := 0;
   incr := 0.1;
begin
   output := 15000 * sin( 10 * sin( input / 300 ) * input )
end.

-------------------------------------------------------------

program test14;
 { synthesis by fm }
var
  a, b;
  frec;
init
  incr := 2 * pi / 22050;
  a := 0;
  b := 0;
  frec := 110;
begin
  output := 20000 * sin( a + b * sin( frec * input ) );
  if input < 8.42 then
    a := a + pi / 4000
  else
    a := a - pi / 3000;
  b := 2 * a;
  frec := frec + 0.008;
end.

-------------------------------------------------------------

program test15;
{  30000 samples }
begin
  output := 5000 * sin( 0.5 * input * sin( 40 * input  ) ) *
                      ( 4 + sin( 2 * input ) );
end.

-------------------------------------------------------------
program _square;
{this program demostrates the overtones
 composition of the square wave }

init
  startvalue := 0;

begin
  output := 20000 * (
                     sin(      110 * input )      +
                     sin(  3 * 110 * input ) /  3 +
                     sin(  5 * 110 * input ) /  5 +
                     sin(  7 * 110 * input ) /  7 +
                     sin(  9 * 110 * input ) /  9 +
                     sin( 11 * 110 * input ) / 11 +
                     sin( 13 * 110 * input ) / 13
                    );
```

```
end.

-------------------------------------------------------------

program ringphone;
{ this program creates a modern telephone sound }

init
  startvalue := 0;

begin
  if input < 0.47 then
    output := 20000 * square( 533 * input )
  else
   begin
     output := 20000 * square( 669 * input );
     if input > 0.94 then
       input := 0;
   end;
end.

-------------------------------------------------------------

program testwhile;
var
  a, b;
begin
  a := 6 * input / pi;
  b := 0;
  output := 0;
  while b <= a do
   begin
     b := b + 1;
     output := output + sin( 220 * b * input ) / b;
   end;
  output := 18000 * output;
end.

-------------------------------------------------------------

program testrepeat;
var
  a, b;

begin
  a := 6 * input / pi;
  b := 0;
  output := 0;
  repeat
    b := b + 1;
    output := output + sin( 220 * b * input ) / b;
  until b > a;
  output := 18000 * output;
end.

-------------------------------------------------------------

program testfor;
var
```

```
  a, b;

begin
  a := 6 * input / pi;
  output := 0;
  for b := 1 to a+1 do
    output := output + sin( 220 * b * input ) / b;
  output := 18000 * output;
end.
```

C. ERROR MESSAGES.

Compiler errors

Duplicate identifier.
The identifier has already been declared within the **var** section. You may be trying to redeclare a reserved word.

Unknown identifier.
This identifier has not been declared.

Type mismatch.
This is due to:

- ♦Incompatible types of the variable and the expression in an assignment statement.
- ♦Incompatible types of operands in an expression.

Overflow in code segment.
Source code too large. This is a very rare situation.

“.” expected.

“)” expected.

“,” expected.

“;” expected.

Identifier expected.
An identifier was expected at this point. You may be trying to redeclare a reserved word.

END expected.

BEGIN expected.

THEN expected.

DO expected.

TO or DOWNTO expected.

:= expected.

UNTIL expected.

Unknown symbol:(symbol).
An illegal character was found in the source text, like: '@', '$', '%', etc.

Syntax Error.
Any kind of syntax error, for instance: A++B.

Run-time errors

Floating point overflow.
A floating-point operation produced an overflow.

Stack segment overflow.
This error is reported if there are too many declared variables or
a too complex expression (like too many nested parenthesis). Is a very rare situation.

Division by zero.
You are trying to divide by zero.

Invalid floating point operation.
This is due to:
♦The argument passed to the **Sqrt** function was negative.
♦The argument passed to the **Ln** function was zero or negative.

D. NOTE FREQUENCIES AND MIDI NUMBERS

NOTE	MIDI	FREQUENCY
C0	24	32.70
C#0	25	34.65
D0	26	36.71
D#0	27	38.89
E0	28	41.20
F0	29	43.65
F#0	30	46.25
G0	31	49.00
G#0	32	51.91
A0	33	55.00
A#0	34	58.27
B0	35	61.74
C1	36	65.41
C#1	37	69.30
D1	38	73.42
D#1	39	77.78
E1	40	82.41
F1	41	87.31
F#1	42	92.50
G1	43	98.00
G#1	44	103.83
A1	45	110.00
A#1	46	116.54
B1	47	123.47
C2	48	130.81
C#2	49	138.59
D2	50	146.83
D#2	51	155.56
E2	52	164.81
F2	53	174.61
F#2	54	185.00
G2	55	196.00
G#2	56	207.65
A2	57	220.00
A#2	58	233.08
B2	59	246.94
C3	60	261.63
C#3	61	277.18
D3	62	293.66
D#3	63	311.13
E3	64	329.63
F3	65	349.23
F#3	66	369.99
G3	67	392.00
G#3	68	415.30
A3	69	440.00
A#3	70	466.16
B3	71	493.88
C4	72	523.25
C#4	73	554.37
D4	74	587.33
D#4	75	622.25
E4	76	659.26
F4	77	698.46
F#4	78	739.99
G4	79	783.99
G#4	80	830.61
A4	81	880.00
A#4	82	932.33
B4	83	987.77
C5	84	1046.50
C#5	85	1108.73
D5	86	1174.66
D#5	87	1244.51
E5	88	1318.51
F5	89	1396.91
F#5	90	1479.98
G5	91	1567.98
G#5	92	1661.22
A5	93	1760.00
A#5	94	1864.66
B5	95	1975.53
C6	96	2093.00
C#6	97	2217.46
D6	98	2349.32
D#6	99	2489.02
E6	100	2637.02
F6	101	2793.83
F#6	102	2959.96
G6	103	3135.96
G#6	104	3322.44
A6	105	3520.00
A#6	106	3729.31
B6	107	3951.07
C7	108	4186.01
C#7	109	4434.92
D7	110	4698.64
D#7	111	4978.03
E7	112	5274.04
F7	113	5587.65
F#7	114	5919.91
G7	115	6271.93
G#7	116	6644.88
A7	117	7040.00
A#7	118	7458.62
B7	119	7902.13
C8	120	8372.02
C#8	121	8869.85
D8	122	9397.27
D#8	123	9956.06
E8	124	10548.08
F8	125	11175.30
F#8	126	11839.82
G8	127	12543.86
G#8	---	13289.75
A8	---	14080.00
A#8	---	14917.24
B8	---	15804.27