COMPUTER SYSTEMS LABORATORY

DEPARTMENTS OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
STANFORD UNIVERSITY * STANFORD, CA 94305

SILT:

A VLSI Design Language

Tom Davis and Jim Clark

Technical Report No. 226

October 1982

This research was supported by the Defense Advanced Research Projects Agency under contract # MDA903-79-C-0680.

SILT:

A VLSI Design Language

Tom Davis and Jim Clark

Technical Report No. 226

October 1982

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract

SILT is an efficient, medium-level language to describe VLSI layout. Layout features are described in terms of a coordinate system based on the concept of relative geometry. SILT provides hierarchical cell description, a library format for parameterizcd cells with defaults for the parameters, constraint checking (but not enforcement), and some name control. It is designed to be used with a graphical interface, but can be used by itself.

Key Words and Phrases: VLSI, design tools, layout editors, relative geometry

© Copyright 1982, Tom Davis and Jim Clark

Table of Contents

List of Figures

1. Introduction

1.1 SILT Overview

In many endeavors, especially VLSI design, what seems to be the last 10% of the work often takes 90% of the time. It is often not too difficult to lay out the initial circuit, but "small" modifications can take an enormous amount of time. One of the main goals of the SILT VLSI layout language is to make such modifications easier. Another goal is to provide a convenient form for general library cells that provides for some "stretch" in the cells. Finally, SILT's naming conventions are designed help the user keep track of the names involved in a large hierarchical circuit.

It is easiest to describe SILT as a language by analogy with programming languages. CIF corresponds to machine language, CLL (a Stanford language that is essentially CIF with symbolic names instead of numbers, see [5]) corresponds to an absolute assembler, and SILT corresponds to a relocatable assembler. SILT is not a "silicon compiler" in that it is descriptive rather than procedural. Because of this, it is not too hard for the user to figure out exactly what geometry will be produced by a given set of instructions. Geometry produced by SILT would bear roughly the same relation to the geometry produced by a true silicon compiler that machine code produced by a language assembler would to that produced by a full-blown compiler.

SILT has some features not present in CIF or CLL. The most important is probably the parameterization of symbols which can be extremely important for a library format. SILT's local names and its method of exporting only certain names outside symbols help to control the size of the name space. Finally, SILT provides some mechanism for constraint checking that is done automatically when symbols are expanded.

SILT's syntax looks much like that of a block structured language such as ALGOL or PASCAL. A SILT file is a series of symbol definitions followed by calls on those symbols. A symbol includes a parameter list, some declarations (including, perhaps, definitions of other symbols), and a series of symbol calls. The symbol calls can be on previously defined symbols or on primitive symbols, such as rectangles, contact cuts, and butting contacts. The scoping rules for names are similar to those in PASCAL, so symbols and variables declared within another symbol are local to it. The mechanism for passing data to and from a symbol will be discussed in greater detail later. There are advantages and disadvantages to this, which are described in 3.1.

SILT can be used by itself to lay out circuits, but it is designed to be used with a graphical front end.

The language is designed to be an interchange form, however, so it in not tied to any particular graphics editor.

It is possible to use SILT using input generated by other graphical editors. The only requirement is that they be able to produce CIF output. In SILT, certain symbols can be declared to be external. A symbol that is so declared can be used in much the same way as other SILT symbols. There are some restrictions, however.

SILT is not a complicated language -- in most cases, the designer should be able to figure out exactly what geometry will be generated by any particular fragment of code. It has no powerful built-in primitives such as “insert 16-bit ALU” or “route signal1<0:15> to signal2<16:31>". The "16-bit ALU” may exist in a library, but the user will have to do the routing for the second example. There are only a few features found in SILT that are not found in some other language -- SILT primarily makes a convenient set of features available within a single language.

1.2 The Implementation

At present, two programs exist for the conversion of SILT to CIF and back again. Both are written in PASCAL. The CIF to SILT converter should be quite portable, but the SILT to CIF program is based on Hennessy’s parser generator (see [2]), so porting the SILT program would also require porting the parser generator as well.

The code is written to run under both TOPS-20 and UNIX. Some minor changes must be made in the code to transfer it from one operating system to another. The source code is the TOPS-20 version, but there are instructions for the edits that must be performed to make the UNIX version at the beginning of the file.

1.3 Using this Document

The easiest way to learn SILT (or any other programming language, for that matter) is by looking at examples. A series of examples is provided in appendix I. The exact syntax for SILT can be found in appendix II, which includes pointers back into the document for discussions of the associated semantics. Finally, there is a list of the SILT reserved words in appendix III.

1.4 Using SILT

SILT currently runs under TOPS-20 (Hedrick's PASCAL) and BERKELEY UNIX (BERKELEY PASCAL). The procedure for using it is similar in both cases. In the TOPS-20 version, when the program starts up, the user will be asked for the name of an input file and an output file. The input file is the SILT source, and the output file will contain the CIF generated. No default extensions are assumed, so the whole file name must be typed in both cases. Any errors encountered are printed out on the terminal. The general philosophy followed by SILT is that it attempts to recover from as many errors as possible. Thus, when the input file contains errors, SILT is not guaranteed to work and run-time errors occasionally occur. The hope is that enough error diagnostics are generated so that the user can correct the errors and try again.

For the UNIX implementation, the standard input and output are used, and most of the errors are recorded in a special file called "errors". A few messages are sent to the terminal. As in the TOPS-20 version, the entire file names must be specified. The standard extensions that most people use are ".slt" for SILT files, and ".cif" for CIF files.

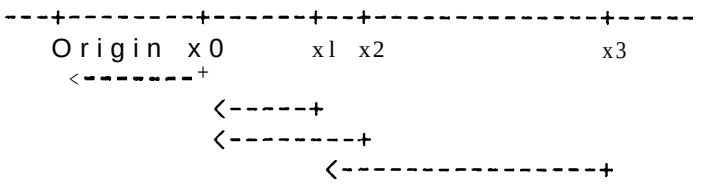
When the UNIX version is used on the VAX, SILT files can be used with the "C" preprocessor. The SILT assembler does not itself call on the preprocessor, but it ignores any lines beginning with the character " # ". The main use of this is for the inclusion of files.

2. Relative Geometry

Each symbol is defined in terms of a coordinate system having an origin and some number of distinguished points on the x- and y- axes. These distinguished points are called x-relative and y-relative points depending upon which axis they fall. (They are simply called relative points, if there is no chance for confusion.) The geometry in a symbol is completely specified in terms of these points. Thus, if x_i and y_i are x- and y- relative points, respectively, then a rectangle (or box) might be defined to have length 2, width x_1-x_0 + 3, and have its lower left corner at (x_0-2, y,). Different values of the x_i and y_i will thus generate rectangles of different shapes and in different positions.

A dependence relation is defined for the set of relative points on each axis. Each point depends on at most one other point. If it depends on no other point, it can be thought of as relative to the origin. In the example below, only x-relative points are considered. x0 is relative to the origin, xl and x2 are relative to x0, and x3 is relative to xl:

Figu re 2- 1: Relative Points



In any instantiation of a symbol, once all the distances between relative points are fixed (the lengths of the arrows above), the geometry is determined. In general, users will not need to specify all **of** these distances; any that are not specified will take default values.

Most rectangles will have each edge a constant offset from a relative point, so changing a single intra-point distance will affect the shapes and positions of only a subset of the rectangles. When a relative point is moved, all geometry ‘that is dependent on that point’s coordinate is moved. In addition, all the relative points dependent on that point are also moved (their distances to the given relative point remain unchanged), so geometry tied to those points is also moved. See the first example in appendix I.

All the geometry in a SILT description is eventually made up of rectangles whose sides are parallel to the coordinate axes. The built-in symbols are of this form, and only rectangles of this form can be specified. Symbols may only be rotated by multiples of 90 degrees.

A graphical front-end for this language can present the symbol on the screen together with the relative points. In addition to the usual commands for adding, deleting, and moving rectangles, the user can add, delete, and move the relative points. When a relative point is moved, all associated geometry is adjusted. The graphical editor must also include commands to associate edges of rectangles with relative points.

In fact, two graphical SILT-based editors have been written. The first was purely experimental, called ALE (see [4]), and was used to experiment with various techniques to interract graphically with SILT constructs. Later, based upon the ALE experiences, another editor, called YALE (see [1]) has been implemented on the SUN workstations. YALE does not implement all of the features of SILT, but does implement the more important ones. All of YALE's input and output is done in a subset of SILT.

3. Names and Data Types

3.1 Names and Scoping

All user-defined names in SILT begin with a letter and are followed by any number of letters, digits, or the underscore character ("_" = ASCII 137B). No case distinction is made, so "AbC5" and "abC5" represent the same name. Users may not use any of the SILT reserved words, which are listed in appendix III.

SILT's scoping rules are similar to those in PASCAL, with one important exception. If SILT's symbols are thought of as PASCAL procedures or functions, then the variables visible with a given nesting of symbols would be the same ones visible within the same nesting of PASCAL procedures. The exception is SILT's export mechanism. This allows a symbol to make a certain set of internally defined symbols visible outside it. This topic is fully discussed in section 3.5.

Future versions of SILT may not allow such complete freedom in nesting functions and in the scope of names. For efficiency, every time SILT expands a symbol, it keeps track of the parameters passed to the symbol and to the CIF code produced. Every time the symbol is expanded, a check is made to see if it has been expanded before with the same parameters. If so, no expansion is done, and a pointer to the already expanded symbol is returned. If a symbol depends on variables not in the parameter list, errors will occur. Future versions of SILT may not allow nesting of functions, and will thus have only two kinds of variables -- local and global. Thus, it is not recommended that full advantage be taken of SILT's nesting mechanism.

3.2 Variable Types

SILT deals with four basic kinds of variables: x- and y-relative points (described in the previous chapter), scalar values and signals. These are declared as xvar, yvar, scalar, and signal, respectively. Variables of type xvar, yvar and signal are simply stored as a single real number, but signals are more complicated. A signal is a collection of triplets, where each triplet consists of an x- and y- coordinate together with an optional process layer (metal, diffusion, polysilicon, etc.).

Scalars are meant as a catch-all to include such things as iteration variables, pull-up ratios, and power requirements. Scalars are not altered when a symbol is transformed (x-relative points are changed to y-relative points when the symbol is rotated 90 degrees).

Signals are meant to be used to identify particular points of the geometry so that various constraints can be checked. A user may, for example, insist that point “a” of signal "b" be connected to point "c" of signal "d" (see 5.12). Signals are transformed as a pair of relative points when their symbol is transformed. Vectors of signals such as "addr<0:23>" can be declared, and one can refer to the components of particular signals with expressions like "a_in.x" or "addr<0>.y". The reason that a signal consists of possibly more than one triplet of values is that the same electrical signal is often available at different points within the symbol. When one asks to have one signal connected to another, it does not matter to which of these points a connection is made. This is especially useful for symbols that have a bus passing through them, and hence each signal on the bus will be available at both ends of the symbol.

All the variables described above can be combined in the usual way with arithmetic operations and constants. SILT makes no checks to ensure that the expressions formed make sense -- it will cheerfully allow the user to multiply relative points or to subtract an x-relative point from a y-relative point.

All numbers in SILT are stored internally as real values, and thus division does not round. SILT also has the integer functions “div” and “mod”. "x mod y" is evaluated as "float(round(x) mod round(y))“, and “div” is handled similarly. Following are a few examples of valid SILT arithmetic expressions:

```
abc + (power*width)
(q mod r) - xyz / (7.3*((p+q) div r))
```

In addition to " + ", "-", "*", "/", "div", and “mod”, SILT includes an (experimental) function “bitop” (standing for “bit operation”). It is a function of two integers (which are produced by rounding as for “div” and “mod”, above), and allows one to determine whether a given bit in the binary expansion of a number is "1" or "0". "bitop(number, bitposition)” yields the numeric value 1 or 0. The least significant bit is number zero, so "bitop(5,0)" yields 1, "bitop(5,1)" is 0, and so on.

In a sense, SILT also deals with boolean values, although there is no way to save such a value in a variable. Arbitrarily complicated boolean expressions like “a > = b AND ((NOT b> = 7) OR (c + e < = d))” can be formed and used in both conditional statements (see section 5.8) or constraints (see section 4.5). At present, the comparitors ">" and "<" are not available because of some restrictions of the parser, but they can be implemented using “NOT” with "> = " and "< = ".

3.3 Symbol Names

All the geometry in a circuit is defined in terms of symbols, the bodies of which are made up of primitive calls and calls on other symbols- Symbols may be defined within other symbols, and symbols so defined are “local” to their containing symbol. Locals within different symbols may have the same names.

The entire SILT file can be thought of as a distinguished symbol that is different from other symbols in that it has no parameters and that it is automatically expanded once at (0, 0). The “file” in SILT is to its symbols much as the “program” in PASCAL is to its procedures.

3.4 Instance Names

A single symbol can be instantiated many times, and any number of particular instantiations may be named. It is common to lay out one- or two-dimensional arrays of symbols, so SILT allows array-like instance names. A symbol call is named if it is preceded by an instance name followed by "::". For example, if "foo" has been declared as a symbol, then it might be instantiated in the following ways:

```
place foo() at . . .              (* no instance name here *)

a:: place foo() at . . .

b[2]:: place foo() at . . .

for i := 0 to 7 do
  for j := 0 to 7 do
    c[i,j]:: place foo() at . . .
```

In arrays of instances, the indices are rounded to the nearest whole number(s). No instance names (including arrays of instances) need to be declared ahead of time. An array of instances must all correspond to the same symbol, although the symbol in question may be called with different parameters in each instance.

3.5 Exports

All variables defined within a symbol are local to the symbol, unless they are specifically exported using an export declaration. Information exported from a symbol is visible only within the symbol that called it. Thus only information that is in some sense important is visible outside a symbol. If symbol “a” calls symbol "b" and symbol "b" calls symbol "c", then the exports of symbol "c" are not automatically visible in the body of symbol “a” unless "b" also specifically exports them. The example in Figure 3-O below illustrates the code required to make "c"'s export visible inside symbol “a” (as b_inst.c_export):

The types of things that are exported often include some subset of the relative points, certain connection points (signals), power requirements, and information about the size of the cell. The size of the cell is automatically exported even if the user does not specifically ask for it (“xmin”, “xmax”, "ymin" and "ymax" are the values automatically exported). The example that follows illustrates the method for accessing exported names. If a symbol is placed without an instance name, its exports are not accessible.

Figure 3-1: Exporting values

```
symbol c();
scalar c_export;
export c_export;
...
symbol b()
scalar c_export;
export c_export;
    begin
    c_inst:: place c() at (0, 0);
    c_export=  c_inst.c_export;
    ...
    end;
symbol a()
    begin
    b_inst:: place b() at (0, 0);
    ...
    end;
...
```

3.6 CIF Names

SILT also has a provision for passing certain names to the CIF file produced. These names can be attached to a point and optionally to a layer. See section 5.9.

4. Symbol Declaration

Every SILT symbol (including the entire SILT file itself) has a header, a series (possibly empty) of declarations, and then a list of symbol commands. This chapter discusses the header and declarations of a symbol definition.

SILT input is free-form in the spirit of PASCAL. Spaces, tabs, and carriage-returns can appear anywhere except within an identifier or a reserved word.

Comments can be imbedded anywhere in a SILT file where a space could appear, and are made up of arbitrary text, surrounded by "(*" and " *)". Comments may be nested, making it possible to comment out arbitrary chunks of SILT code.

All symbols must be defined before they are used -- there is nothing that would correspond to a PASCAL “forward” declaration. Instance names need not be declared ahead of time, but all other variables must be.

4.1 Symbol Parameters

It is possible to pass any number of relative point values to a symbol via the parameter list. SILT also allows the user to pass other values to a symbol, such as scalars representing power requirements or pullup ratios. Each parameter should have a default value associated with it that is usually chosen to produce a cell-of minimum size. See section **4.4.** Since the minimum cell is usually what is desired, parameter specification will be the exception rather than the rule. For this reason, both key calls and positional calls are implemented in SILT. If the symbol foo has a header that looks like:

```
symbol foo(xvar x0, x1, x2);
```

Then the following two forms are valid calls on the symbol:

```
place foo(x0=3, x2=15) at (0, 0);
place foo(3, 4, 18) at (0, 0);
```

In the first case, xl will get the default value, whatever it is. In the second, x0, xl, and x2 would be assigned to 3, 4, and 18, respectively.

4.2 Declaration of Variables

The symbol header is followed by an optional constant declaration and then a series of variable declarations- Constants serve the same purpose as constants in PASCAL, and their values cannot change after their initial declaration. A constant declaration might look something like:

```
constant a := 1; b := 5; c3po := 11.38;
```

Any variables that are to be local'to the symbol are declared next as in the following example:

```
xvar x0, x1, x2;
yvar y0;
scalar power, indx;
signal a, addr<0:23>, vdd, gnd;
```

The variable declarations may appear in any order.

4.3 Exports

Any variable declared within a symbol can be exported by including its name in an export declaration. In addition, variables from the parameter list can also be exported. (This last is not so silly as it seems -- since various defaults may be taken, the user would otherwise have no easy way to find out some of the relative point positions.) As was stated before, every symbol effectively includes the following:

```
xvar xmin, xmax;
yvar ymin, ymax;
export xmin, xmax, ymin, ymax;
```

The general export declaration looks exactly like the example above. If a signal vector is exported, only its name is necessary -- the size has already been declared. The following example illustrates all the possibilities:

```
symbol foo(yvar y9);
xvar a, b;
signal c, d<0:5>, e;
export a, c, d, b, y9;
```

4.4 Defaults

A default declaration is made up of a list of assignment statements that are interpreted in a special way. The left-hand-side (LHS) of an assignment must be a member of the formal parameter list, but the right-hand-side (RHS) can be an arbitrary expression. The only constraint is that all variables in the RHS must be known by the time the expression is evaluated. A typical set of default values (in this case for Figure 3-O above) will look something like this:

```
default x0 := 9; x1 := x0 + 6; x2 := x0 + 9; x3 := x1 + 17;
```

The form above makes the relative relations among the points clear -- x0 is relative to the origin, xl is relative to x0, and so on.

SILT expands a symbol's parameter list by going through the default declarations in order, and if the LHS variable is unknown, then the RHS is evaluated and assigned to it. If the LHS is known, SILT advances to the next assignment in the default list. If the symbol having the default declaration in the last paragraph were called with x0 = 8 and x2 = 11, then the symbol would be expanded with x0 = 8, xl = 14, x2 = 11, and x3 = 31.

4.5 Constraint Declarations

It is possible to declare within any symbol a set of constraints to be checked when the cell is instantiated. A constraint is entered as two arbitrary expressions separated by "< = " (less-than-or-equal), "> = ", "<>" (not-equal) or " = ". Various information about design rules, power requirements, or anything else can be included. SILT makes no attempt to force the constraints to be satisfied -- it simply warns the user of a possible error if a constraint violation occurs.

All the boolean expressions in the constraint list are evaluated after the default list has been processed. If errors are discovered, the user is warned, but the SILT assembly continues. SILT makes no attempt to stretch or shrink geometry to force the constraints to be satisfied -- it merely prints a warning if there is an error.

4.6 Symbol Definitions

After any combination of the above declarations, any number of local symbols may be declared. The program fragment below gives the form of a complete symbol definition, where the "..." is replaced by any number of symbol commands which will be discussed in chapter 5.

```
symbol foo(xvar x0, yvar y0, scalar bar);
constant const1 := 7;
xvar x_loc;
signal sig<0:7>;
export x0, x_loc, sig;
default x0 := 5; y0 := 7;
        scalar := x0 - y0 + 2; (* legal, but senseless *)
    begin
    ...
    end;
```

5. The Symbol Body

A variety of commands are available within the body of a SILT symbol. Each is discussed in detail in the following sections.

5.1 Primitive Calls

At present, there are three types of primitive calls available in SILT. These include contact cuts, butting contacts, and boxes (rectangles).. The calling sequence for each of the above types is similar to the calling sequence for general symbols, but there are a few restrictions. In most of the examples in this document, the word “place” has been put in front of each call on either primitive or user-defined symbols. It is optional, and simply serves to make the SILT text easier to read.

Without too much difficulty, it is possible to add other primitive symbol types to the language, should they prove to be important.

5.1 .1 Box Calls

The following three examples illustrate possible box calls:

```
place box(1, 4) at (x0+5, y7+wirewidth/2);
place box (x1-x0, 4, poly) at (0, -23);
box (1,4) at (4, 5); (* "place" is not required *)
```

The first two parameters enclosed in parentheses after “box” are the x-length and y-length. If there is a third parameter, it must be a layer in the set {metal, poly, diff, implant, contact, buried}. It indicates the layer upon which a particular box is to be placed. If no layer is indicated, then the default layer (from the “with” command -- see section 5.7) is used. If the box command appears without a layer specification, and is not within the scope of a with command, then an error occurs.

The last pair of numbers are the x- and y- coordinates of the lower left corner of the box. None of the other transformations (described later) may be applied to a box call.

Box calls cannot be given an instance name.

An alternative form of the box command can be used that substitutes the word “to” for “at”. A typical call might look something like this:

```
place box (1, y3+17, poly) to (7, y3+22);
```

In this form,. the first pair of numbers serve as coordinates for the lower-left corner and the second

two as coordinates for the upper-right corner. This form makes it much easier to see exactly which reference point is associated with each edge, and hence provides a much better graphical interface.

5.1.2 Contact Cuts

Contact cuts come in two flavors -- one connects metal to poly, and the other connects metal to diffusion. A contact cut may be placed as follows:

```
place cut(poly) at (3, 6);
place cut(diff) at (3, 6);
place cut() at (0, 0);
place cut at (0,0);
```

In the third and fourth examples, the layer is chosen in the same way as it is for box calls. If the layer is not in {poly, diff} then an error occurs.

The cut symbol has its origin at the lower-left corner. It can be rotated and flipped, but it is not really necessary for this symbol. See section 5.2.

Contact cuts calls cannot be given an instance name.

5.1.3 Butting Contacts

The butting contact symbol has only one type -- the standard Mead-Conway butting contact (see [3]) oriented horizontally with the diffusion on the left. The origin is at the lower-left corner, and when the cell is placed, any standard transformation can be applied to it. (See the next section for a discussion of transformations.)

Since there is only one type of butting contact, the calls must look like one of the following two examples:

```
place butt() <transformation>;
place butt <transformation>;
```

Like contact cuts and boxes, butting contact calls cannot be given an instance name.

5.2 Transformations

Every symbol that is either user-defined or built-in (butting contact, contact cut) has an origin at the point (0, 0). For the built-in symbols, the origin happens to be at the lower left-hand corner, but this need not be the case for user-defined symbols.

Three different kinds of transformations are allowed: translation, rotation, and reflection. Rotation always takes place about the origin and reflection through either the x- or y- axis. Since all geometry is constrained to be parallel to the coordinate axes, rotations can only occur in multiples of 90 degrees.

Any sequence of translations can be applied to a symbol, applied in the order in which they appear in the SILT description (left to right). The order is important -- a rotation followed by a translation is much different from the same translation followed by the rotation.

Rotations are defined relative to a standard clock face (this idea comes from the CLL language). Imagine the origin of the symbol at the center of a clock face with an arrow super-imposed on it pointing to 12 o'clock. A rotation of 3 leaves the origin in place, but rotates the arrow so that it points to 3 o'clock, and so on. The only rotations allowed are 3, 6, and 9. The syntax for a rotate command is something like "rotated 3".

Reflections are either up-down (up-down means along the y-axis, or across the x-axis) or left-right. The syntax for a reflection must be one of: "flipped ud", "flipped lr" or "flipped rl". The last two are equivalent.

A translation has the form "at (x-trans, y_trans)". This has the effect of translating the origin of the symbol to the point (x-trans, y_trans).

A complete transformation is made up of any number (including zero) of the above, optionally separated by commas. If no transformation is given, the symbol is placed at (0,0) in the standard orientation. Supposing that the symbol "foo" has already been declared, the following are valid symbol calls on "foo":

```
place foo() at (0, x0+7);
place foo() flipped ud, rotated 3 at(0,0);
place foo() at (5, -3) flipped ud;
foo at (4, 6);
abc:: foo;
fooinst:: place foo(power=3) flipped ud at (0, 5) rotated 9;
place foo() rotated 3;
place foo() at (3, 4) at (7,5);
```

The next to last example above places the rotated cell at the origin, and the last one is equivalent to:

```
place foo() at (10, 9);
```

5.3 Symbol Calls

A call on a user-defined symbol must appear after the symbol has been defined. If desired, the call may be preceded by an instance name, followed by "::". Only labelled instances can have their exports referred to later.

The parameter list is optional, and if it is omitted, all the default values will be taken. Parameters may be specified either by a positional call or a key call. If it is a positional call, the parameters are listed in the order in which they appeared in the symbol declaration. If it is a key call, the form is:

<formal parameter name> "=" <actual parameter>

All calls are by value, not by reference. Following is a short list of examples of symbol calls. Assume that the symbol "foo" has already been declared:

```
a:: place foo(x1=5, y3=7) at (2, 3);
b[i+1]:: place foo(x1=x1, power=13) flipped ud at (3, -5);
q[i,j]:: place foo(x1=i*3) at (x0+q[i-1,j].xmax, 17);
place foo(in<0:3> = x<0:3>);
place foo(1, 2, 3) at (3, 4);
```

In the second example in “xl = xl ", the “xl” on the left is the name of the parameter in "foo", and the “xl” on the right is the value of “xl” in the calling symbol. It is impossible to tell (from the example) what variables get set to what values in the final example, since we do not know the order in which the parameters were declared. See also section 4.1.

5.4 The Block Command

The block command is just a convenient way to group together a number of SILT commands for the benefit of a with command, an iteration command, or a conditional command. The syntax is “begin”, followed by any number of SILT commands, followed by “end;“. SILT blocks can be nested, if desired. An example follows:

```
with poly do
    begin
    place box (1,2) at (3,4);
    place box (3,4,diff) at (5,6);
    place box (7,8) at (9, 10)
    end;
```

[1] A warning is in order here. This is a moderately unusual construction, but there must be a space between the ">" and the " = ". Otherwise, the lexical scanner will interpret it as ">= ".

5.5 Iteration Commands

SILT contains a simple iteration scheme that behaves almost exactly like a simple form of the PASCAL “for” statement. The following examples give most of the flavor of the command:

```
for i := 1 to 8 do
    for j := 1 to 8 do
        begin
        place box(1,1,metal) at (2*i, 2*j);
        place box(1,1,poly) at (2*i+1, 2*j)
        end;
```

In this first example, "i" and "j" must be declared (presumably as scalars, although they could be of type xvar or yvar).

```
a[0]:: place foo(param=0) at (x0, 0);
for i := 1 to 7 do
    a[i]:: place foo(param=i) at (a[i-1].xmax, 0);
```

The example above places 8 copies of the symbol foo side by side. (In this example, "foo" is assumed to have xmin = 0.) The first instance of "foo" must be placed outside the “for” statement so that the following instances can each refer to the instance to the left. All the instances could have different widths, depending upon what "foo" does with "param".

If the intent is simply to place an array of identical symbols in a linear or rectangular array, use the array command, described in the next section.

5.6 The Array Command

SILT is a rich language, and it would be difficult to implement a graphical front-end that is capable of taking advantage of all SILT’s features. One of the more difficult features to implement in its full generality is the iteration command discussed in the last section. Since one of the most common things to do in VLSI design is to lay out an array of symbols, the array commands provide a restricted form of iteration that can fully implemented by a graphical system.

The array command allows the user to place a linear or rectangular array of symbols at a given starting point with a given spacing. All instances of the symbol must be identical. If no spacing is specified, the symbols are placed with x-separation xmax - xmin, and y-separation ymax - ymin. Some examples of the array command follow:

```
place array a[0..7] of foo at (q7, b3+9);
array blat[0..3, 0..5] of foo(6,3-delta,8) at (43, ypos-6);
array bar[0..20] of foo() spacing (5,8) at (11,5);
```

As in other placement commands, the keyword “place” is optional. The array dimensions are described PASCAL-style, and any sort of symbol call can be used after the keyword “of”. If “spacing” appears, it is followed by a delta-x, delta-y pair, and the point following the “at” gives the coordinates of the symbol having the smallest x (and y, if there is one) coordinate. It is recommended that this form be used instead of general iteration when it is possible so that the SILT generated can be more easily handled by a graphics system.

The array command:

```
place array a[0..7] of foo at (0,0);
```

is exactly equivalent to the command:

```
for i := 0 to 7 do
    a[i]:: place foo at (0,0);
```

except that the variable "i" is not present. One can, however, refer to such things as "a[3].xmin" and "a[5].foo_output" in the usual way.

5.7 The With Command

The “with” command sets the default layer for one SILT command. That command may, of course, be a block, so the “with” can extend over any number of statements. In the example in section 5.4, the first and last boxes are placed in poly. If a box call has a layer specification, it holds only for that box. “With” commands can be nested, with the following results:

```
with poly do
    begin
    place box(1,1) at (2,2);            (* set in polysilicon *)
    with metal do
        begin
        place box(1,1) at (3,5);        (* set in metal *)
        place box(1,2,diff) at (2,3);   (* set in diffusion *)
        place box(1,1) at (10, 11);     (* set in metal *)
        end;
    place box(5, 9) at (19, 20)         (* set in polysilicon *)
    end;
```

Allowable layers for the “with” command include: “buried” (buried contact), “contact” (contact cut), "diff" (diffusion), “implant” (implant), “metal” (metal) and "poly" (polysilicon).

5.8 Conditional Commands

A SILT conditional command is a simple “if-then” statement. It takes the general form:

if <boolean expression> then <SILT command>;

The boolean expression is evaluated, and the SILT command is expanded if it is true. There is no

“else” clause -- use another conditional statement if this is necessary. This is not intended to be a heavily-used feature of SILT. It could be used to generate river-routing cells, for example, and to decide which way a wire bends.

5.9 The CIF_List Command

The "CIF_List" command attaches a name to a point (and optionally to a layer as well) in the CIF file generated. The CIF generated by this command is not standard CIF -- it is the “94” user extension used at Stanford and some other sites. The text is listed in the plot, and at Stanford, at least, it is required to be a single identifier (no spaces allowed). Examples of two typical CIF_List commands follow:

```
cif_list "text" at (9, 10);
cif_list "text1" at (10, 11, metal);
```

5.10 External Declarations

If libraries of symbols in CIF form are available, it is possible to make use of the symbols contained in them in SILT using the “extern” command. The symbol that is declared external is assumed to be absolute, and information about its minimum and maximum x- and y-values is not available. If SILT encounters an Extern statement, an entry is made in a symbol file that contains the CIF number used internally, the symbol name (used in the CIF file), and the file name. Extern declarations should be intermixed with the rest of the symbol declarations in the SILT file. A typical extern declaration appears below:

```
extern cifname "filename";
```

The information in the symbol file can be used to link together SILT and CIF files, either manually, or with a program. At present, there is a linker that can link a single SILT file to any number of ICARUS-produced CIF files.

5.11 Assignment Statements

There are a number of allowable types of assignment statements allowed in SILT. The right-hand-side is evaluated, and is’ assigned to the variable on the left-hand-side. An error occurs if the two sides do not conform. Any variable that is stored as a single real number conforms to any other, any signal conforms to any other, and any signal vector of length n corresponds to any other of the same length, and so on. Following are a few examples of typical SILT assignment statements:

```
a := 5;
sig.x := 7 + a;
sig1 := sig2;
sigvec<0:3> := sigvec1<4:7>;
sig1 := metal;
sig1<2>.y := 15;
sigvec<0:7> := inst.sigout<0:7>;
sig<5> := (1,2,metal), (3, 4), (5, 6, diff);
```

All the examples above except for the last should be clear. In the last case, the fifth signal in the vector "sig" is assigned a series of point-layer combinations. These are added to any sets of values the signal may already have. If an assignment is made to a signal suffixed with a ".x" or ".y", or a layer assignment is made, then if the signal has a point defined, its corresponding value will be replaced. If it has no value, a new slot is made. Thus, one can put in a new signal value as follows:

```
sig.x := 5;
sig.y := 7;
sig := poly;
```

But if the command:

```
sig.x := 6;
```

is given, a new signal point is not begun -- the value 5 is simply written over.

Assignments of the form:

```
sig := (3,4,poly);
```

always generate a new point instance.

It is legal, although probably bad practice, to reuse variable names as illustrated in the following example:

```
i := 7;
place box(i,i) at (0,0);
i := 9;
place box(i,i) at (10,10);
```

If assignments occur only once to each variable, the language becomes declarative. Future versions of SILT may print warnings when a variable (other than an iteration variable, of course) is assigned to more than once.

5.12 Connect Commands

The SILT connect commands are used to make sure that a signal point placed by one symbol call coincides with a similar point in another symbol call. If a connect command is given, SILT simply makes sure that the points in question do coincide. If not, an error message is generated, and SILT continues to expand the file. Remember that a signal can correspond to many points. If two signals are connected (that is, an appropriate connect command appears in the SILT file), the connection is

considered to be successful if any point from one coincides with any point from the other. If layers are specified as part of the point, then the layers must conform as well. SILT allows certain combinations of layers for this purpose. metal-poly, for example, means that this point can be connected to another point of type metal or poly.

Following are some simple examples of the use of the connect command:

```
connect a to b;
connect a.sigout to b.sigout;
for i := 1 to 7 do
    for j := 0 to 7 do
        connect a[i].sigout<j> to a[i-1].sigin<j>;
```

In addition to the simple connect command illustrated above that connects a signal to a signal, there is a sometimes more convenient form that makes sure that all the signals having the same name in two instances are connected. This is useful if a bus passes through, and one would like to make sure that all the bus signals are connected. Some examples follow:

```
connect all inst1 to inst2;
for i := 1 to 7 do
    connect all inst[i] to inst[i-1];
```

5.13 The Route Command

The route command implements a simple river-router to connect one signal vector of points to another. The points in the signals must include a layer chosen from (diff, poly, metal}. Both signal vectors must be essentially parallel -- i.e. they must both be monotonic in the x-coordinates or both in the y-coordinates. Some examples of the route command appear below:

```
route(sig1<0:7>, sig2<0:7>, 3, ud);
route(sig1<0:7>, sig2<8:15>, 4, lr);
```

In the first example, sigl<O> is routed to sig2<0>, and so on; in the second example, sig1<0> is routed to sig2<8>. In the first example, the widths of all the routing wires are 3, and in the second example, 4. The first example routes the wires generally up-down (the signal vectors are parallel to the x-axis), and in the second example, the routing is from right to left. The endpoints of the wires do not have to lie in a line. The points in the signal vectors mark the centers of the endpoints of the wires, and the wire layer is determined by the layers of the endpoints. There is an example of the use of the route command in l.

I. Examples

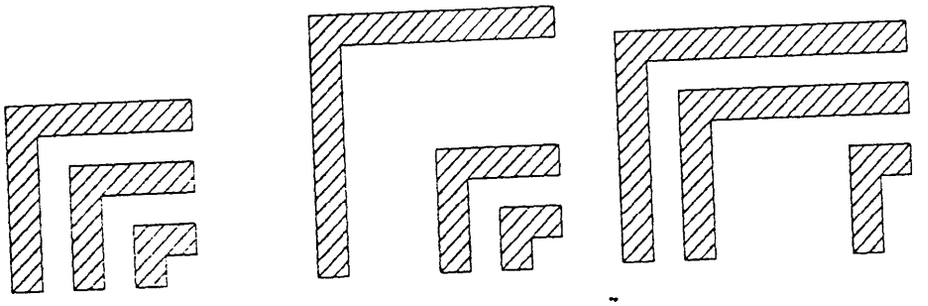
Three examples appear below. Some of them are not done in the most efficient way, but are done in a way that illustrates as many features of SILT as possible. All the examples are probably too small to be realistic.

The first one is extremely simple, and is made up of a few calls on a symbol that consists of a few boxes. It is intended to illustrate relative geometry. The second example illustrates the use of the route command. It also illustrates a few other features of SILT.

The third is the inverter from the Mead-Conway text [3] with some stretch built into it. In this example, the metal-metal distance can be altered, the input and output wires can be shifted up and down, and the pulldown ratio can be altered. Four different cell configurations are illustrated.

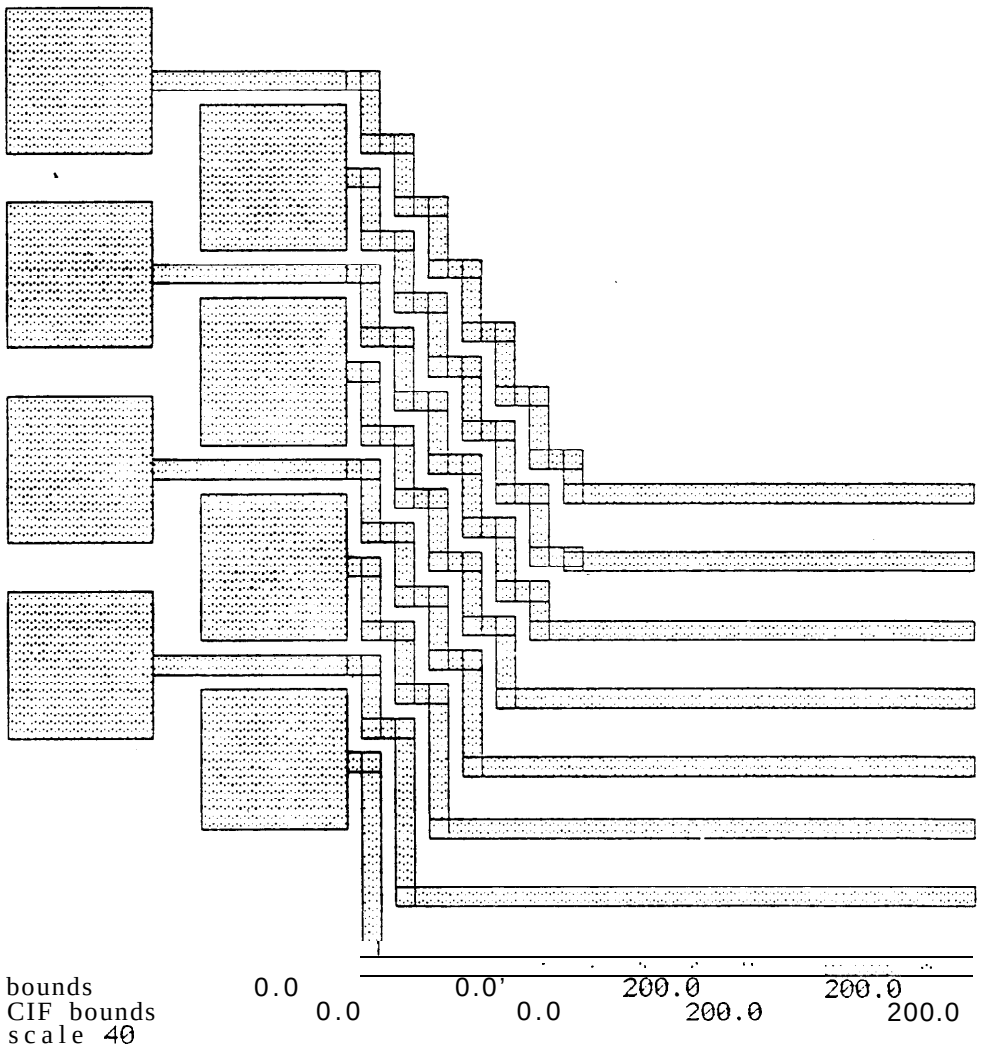
```
file relative_geometry;
symbol three_bend(xvar x1, x2, x3; yvar y1, y2, y3);
default x1 := 4; x2 := x1 + 4; x3 := x2 + 4;
        y1 := 2; y2 := y1 + 4; y3 := y2 + 4;
    begin
    with poly do
        begin
        place box(2, y3+2) at (0, 0);
        place box(2, y2+2) at (xl, 0);
        place box(2, y1+2) at (x2, 0);
        place box(x3 - x2 - 2, 2) at (x2+2, y1);
        place box(x3 - x1 - 2, 2) at (x1+2, y2);
        place box(x3 - 2, 2) at (2, y3)
        end;
    end;
begin
place three_bend at (0,0);
place three_bend(x1 = 8, y3 = 15) at (20, 0);
place three_bend(x2 = 15, y1 = 5) at (40, 0)
end.
```

Figure 5-1: Relative Geometry Example



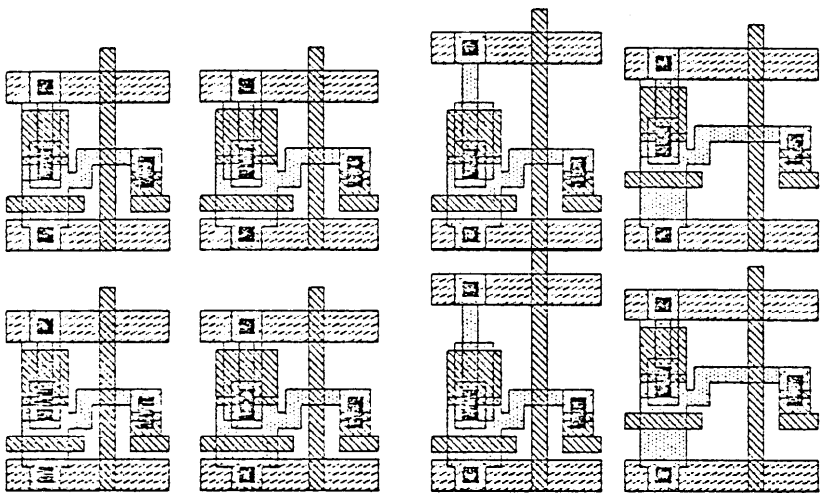
```
file pad_router;
constant pad_size := 30;
scalar i;
signal wire_target<0:7>;
symbol pad();
signal output;
export output;
    begin
    place box(pad_size, pad_size, metal) at (0,0);
    output := (pad_size/2, pad_size, metal)
    end;
symbol pad_raft(scalar pad_spacing);
signal pad_outputs<0:7>;
scalar i, pad_sep;
export pad_outputs;
    begin
    pad_sep := pad_size + pad_spacing;
    for i := 0 to 3 do
        begin
        lower[i]:: place pad at (i*(pad_sep), 0);
        upper[i]:: place pad at (pad_sep/2+i*(pad_sep),
                                 pad_sep);
        end;
    for i := 0 to 7 do
        begin
        if (i mod 2) = 0 then pad_outputs<i> := lower[i div 2].output;
        if (i mod 2) = 1 then pad_outputs<i> := upper[i div 2].output
        end;
    end;
begin
raft:: place pad_raft(pad_spacing = 10) at (0,0);
for i := 0 to 7 do
    wire_target<i> := (100 + i*14, 200, metal);
route(wire_target<0:7>, raft.pad_outputs<0:7>, 4, ud);
end.
```

Figure 5-2: River Routing Example



```
file m_c_inverter;
symbol inv(xvar x1; yvar y1, y2; scalar p_width);
xvar diff_ctr, diff_edge;
default y1 := 5; y2 := y1 + 14;
        p_width := 6; x1 := 12 + p_width - 6;
constraint p_width >= 6; x1 >= 6 + p_width;
           y1 >= 5; y2 >= y1 + 14;
    begin
    diff_ctr := 2 + p_width/2;
    diff_edge := 2 + p_width;
    place butt rotated 9 at (diff_ctr+2, y1+3);
    place butt rotated 3 at (x1+4, y1+8);
    with diff do
        begin
        place cut at (diff_ctr -2, 0);
        place cut at (diff_ctr - 2, y2);
        place box (p_width, y1+4) at (2,3);
        place box (1,2) at (diff_edge, y1+3);
        place box (2,5) at (diff_edge+1, y1+3);
        place box (x1-diff_edge+1, 2) at (diff_edge+3, y1+6);
        place box (2, y2-y1-7) at (diff_ctr-1, y1+7)
        end;
    with poly do
        begin
        place box (p_width, 7) at (2, y1+6);
        place box (4+p_width, 2) at (0, y1);
        place box (2, y2+7) at (x1, 0);
        place box (5, 2) at (x1+4, y1)
        end;
    with metal do
        begin
        place box (x1+9, 4) at (0,0);
        place box (x1+9, 4) at (0, y2)
        end;
    place box(5, 10, implant) at (diff_ctr-2.5, y1+4.5)
    end;
symbol inv_set();
    begin
    place inv() at (0,0);
    place inv(p_width = 8) at (25,0);
    place inv(p_width = 7, y2=24) at (55,0);
    place inv(x1=16, y1=8) at (80, 0)
    end;
begin
inst:: place inv_set() rotated 3 at (0,0);
place inv_set() rotated 3 at (inst.xmax, 0);
end.
```

Figure 5-3: Inverter Example



II. Syntax

```
expression  ->  term;
     ->  expression  '+'  term;
     ->  expression  '-'  term;

term  ->  factor;
      ->  term  '/'  factor:
      ->  term  '*'  factor:
      ->  term  'div'  factor;
      ->  term  'mod'  factor;

factor  -> unsigned_factor;
        -> '-' unsigned-factor;

unsigned-factor   ->  'number';
                  -> '(' expression  ')';
                  ->  value;
                  ->  'bitop'  '('  expression  ','  expression  ')';

boolexp  ->  boolterm;
         ->  boolexp  'or'  boolterm;

boolterm  ->  boolfactor;
          ->  boolterm  'and'  boolfactor;

bsolfactor  ->  primboolfactor;
            -> 'not'  primboolfactor;

primboolfactor  -> '('  boolexp  ')';
                ->  constraint:

instance  ->  instance-name;
          ->  instance-name  instance-qualifier;

simple_variable  ->  variable-name;
                 ->  variable-name  signal-coordinate;

suffixed_variable  ->  simple-variable  suffix;

suffix  -> '.x';
        -> '.y';

instance-name  -> 'ident';

variable-name  ->  'ident';

value  ->  instance  '.'  simple-variable;
       ->   simple-variable;
       ->  instance  '.'  suffixed-variable;
       ->  suffixed-variable;

signal-coordinate  ->  '<'  expression  '>';
```

```
signal-range -> '<' expression ':' expression '>';

signal-vector -> variable-name signal-range;

instance-qualifier -> '[' expression ']';
                   -> '[' expression ',' expression ']';

symbol-definition -> 'symbol' 'ident' formal_parameter_list ';'
                     symbol-head
                     'begin'
                     symbol-tail
                     'end' ';';
                  -> 'extern' 'ident' 'string' ';';
formal-parameter-list -> '(' variable-list ')';
                      -> '(' ')';
                      -> ;

symbol-head -> constant-declaration
               variable-declaration
               export-declaration
               default-declaration
               constraint-declaration
               symbol-definition-list;

symbol-definition-list -> symbol-definition-list symbol-definition;
                       -> ;

constraint-declaration -> 'constraint' constraint-list ';';
                       -> ;

constraint-list -> boolexp;
                -> constraint-list ';' boolexp;

constraint -> expression"<=' expression;
           -> expression '>=' expression;
           -> expression '<>' expression;
           -> expression '=' expression;

default-declaration -> 'default' default-list ';';
                    -> ;

constant-declaration -> 'constant' constant-list ';';
                     -> ;

constant-list -> constnt;
              -> constant-list ';' constnt;

constnt -> 'ident' ':=' expression;

default-list -> default;
             -> default-list ';' default;

default -> simple-variable ':=' expression;
        -> suffixed-variable ':=' expression;
```

```
variable-declaration  ->  variable-list ';';
                      ->  ;

export-declaration  ->  'export'  ident_list ';';
                    ->  ;

variable-list  ->  variable-type-list;
               ->  variable-list  ';'  variable-type-list;

variable-type-list  ->  'xvar'  ident_list;
                    ->  'yvar' ident_list;
                    ->  'scalar'  ident_list;
                    ->  'signal'  *signal-list;

signal  -> 'ident';
        -> 'ident' signal-range;

signal-list  ->  signal;
             ->  signal-list  ','  signal;

ident_list -> 'ident';
           -> ident_list ',' 'ident';

symbol-tail  ->  symbol-command;
             ->  symbol-command  ';'  symbol-tail;

silt-file  ->  'file'  'ident'  ';'
               symbol-head
               'begin'
               symbol-tail
               'end'  '.';

symbol-command  ->  symbol-label  '::'  unlabeled-command;
                ->  unlabeled-command;
                ->  box-call;
                ->  box-to-call;
                ->  butt-call;
                ->  cut-call;
                ->  connect-command;
                ->  assignment;
                ->  with-command;
                ->  iterate-command;
                ->  cif_list_command;
                ->  block-command;
                ->  conditional-command;
                ->  route-command;
                ->  array_command;
                ->  ;

route-command  ->  'route'  route-list;

route-list  ->  '('  vectored-signal  ','  vectored-signal  ','  expression  ')';
            ->  '('  vectored-signal  ','  vectored-signal  ','
                     expression  ','  route-direction  ')';
```

```
route-direction  -> 'ud';
                 -> 'rl';
                 -> 'lr';

vectored-signal  ->  signal-vector;
                 ->  instance '.' signal-vector;

cif_list_command -> 'cif_list' 'string' 'at' position;

unlabeled-command  ->  symbol-call;

block-command  ->  'begin'
                   symbol-tail
                   'end';

symbol-call  -> 'place' 'ident' actual-parameter-list
                orientation-specification;
             -> 'ident' actual-parameter-list
                orientation-specification:

orientation-specification   ->   transformation-list;
                           -> ;

transformation_list  ->  transformation;
                     ->  transformation ',' transformation_list;
                     ->  transformation  transformation-list;

actual-parameter-list  ->  '(' key-call-list ')';
                       -> '(' position_call_list ')';
                       -> '(' ')';
                       -> ;

key-call-list  ->  key-call;
               ->  key-call-list  ',' key-call;

position-call-list  ->  expression:
                    -> position_call_list ',' expression;

key-call  ->  simple-variable '=' expression;
          ->  signal-vector '=' signal-vector;
          ->  signal-vector '=' instance '.' signal-vector;

position  -> '(' expression ',' expression ')';
          -> '(' expression ',' expression ',' layer ')';

transformation  ->  'flipped' 'ud';
                ->  'flipped' 'rl';
                ->  'flipped' 'lr';
                ->  'rotated'  'number';
                ->  'at'  position;

box-call  ->  'place'  'box'  box-parameters  'at'  position;
          ->  'box'  box-parameters  'at'  position;
```

```
box-to-call  ->  'place'  'box'  box_parameters  'to'  position;
             ->  'box'  box-parameters  'to'  position;

box-parameters  ->  '('  box-size  ')';
                ->  '('  box-size  ','  layer  ')';

box-size  ->  expression  ','  expression;

butt-call  ->  'place'  'butt'  orientation-specification;
           ->  'place'  'butt'  '('  ')'  orientation-specification;
           ->  'butt'  orientation-specification;
           ->  'butt'  '('  ')'  orientation-specification;

cut-call  ->  'place'  'cut'  orientation-specification:
          ->  'place'  'cut'  '('  ')'  orientation-specification;
          ->  'place'  'cut'  layer-parameter   orientation-specification;
          ->  'cut'  orientation-specification;
          ->  'cut'  '('  ')'  orientation-specification;
          ->  'cut'  layer-parameter   orientation-specification;

layer-parameter  ->  '('  'diff'  ')';
                 ->  '('  'poly'  ')';

layer  ->  'poly';
       ->  'metal';
       ->  'diff';
       ->  'buried':
       ->  'implant';
       ->  'contact':
       ->  'metal_poly';
       ->  'diff_poly';
       ->  'diff_metal';
       ->  'none';

assignment  ->  simple-variable  ':='  expression;
            ->  suffixed-variable  ':='  expression;
            ->  simple-variable  ':='  layer;
            ->  simple-variable  ':='  position-list;
            ->  signal-vector  ':='  signal-vector;
            ->  signal-vector  ':='  instance  '.'  signal-vector;

position-list  ->  position;
               ->  position-list  ','  position;

connect-command  ->  'connect'  value  'to'  value:
                 ->  'connect'  'all'  instance  'to'  instance:

with-command  ->  'with'  layer  'do'  block-command;

iterate-command  ->  'for'  'ident'  ':='  expression  'to'
                     expression  'do'  symbol-command;

conditional-command  ->  'if'  boolexp  'then'  symbol-command;

symbol-label  ->  instance;
```

```
array-command  ->  'array'  'ident'  a-list  'of'  a-list-call
                       spacing  'at'  position;
                   ->  'place'  'array'  'ident'  a-list  'of'  a-list-call
                       spacing  'at'  position;

a-list  ->  '['  expression  '..'  expression  ']';
        ->  '['  expression  '..'  expression  ','
                 expression  '..'  expression  ']';

a-list-call  ->  'ident'  actual-parameter-list;

spacing  ->  'spaced'  position:
         ->  ;
```

III. SILT Reserved Words

The following identifiers are reserved by SILT. The only onesthatmay be a little surprising are "x" and "y". Thesa are used as suffixes of points.

ALL	AND	ARRAY	AT	BEGIN
BITOP	BOX	BURIED	BUTT	CIF_LIST
CONNECT	CONSTANT	CONSTRAINT	CONTACT	CUT
DEFAULT	DIFF	DIFF_METAL	DIFF_POLY	DIV
DO	END	EXPORT	EXTERN	FILE
FLIPPED	FOR	GLASS	IF	IMPLANT
LR	METAL	METAL-POLY	MOD	NONE
NOT	OF	OR	PLACE	POLY
RL	ROTATED	SCALAR	SIGNAL	SPACING
SYMBOL	THEN	TO	UD	WI T H
X	XMAX	XM I N	XVAR	Y
YMAX	YMIN	YVAR		

References

1. Davis, Tom, and Clark, Jim. YALE User's Guide: A SILT-Based VLSI Layout Editor. Stanford, 1982.

2. Hennessy, John. The Stanford Parser Generator. Stanford, 1980.

3. Mead, Carver, and Conway, Lynn. *Introduction* to *VLSI* Systems. Addison Wesley, 1980.

4. Newell, Martin. The ALE Graphics Editor. XEROX PARC

5. Saxe, Tim. CLL - A Chip Layout Language (Version 2). Used in the Stanford VLSI design class