EARL:
An Integrated Circuit Design Language

Chris Kingsley

Computer Science
California Institute of Technology

5021:TR:82

Earl: An Integrated Circuit Design Language

Chris Kingsley
Computer Science Department
California Institute of Technology
Pasadena, California 91125

Technical Report #5021

In Partial Fulfillment of the Requirements
for the Degree of Master of Science

The work described in this document was sponsored by the Defense Advanced Research Projects Agency, ARPA order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597

Contents

CHAPTER 1

Earl User's Guide

1. Introduction

Earl is an integrated circuit design system that supports the separated hierarchy design methodology. In it, one designs leaf cells, that have only mask layout in them, and composition cells, that have only other cells in them. Earl provides stretchable cells and guarantees that the connection points of the cells connect in the finished layout. It is an interpreted language that was designed for simplicity, and to be able to easily express operations that are useful for circuit design.

Earl has functions for defining cells and connection points (called ports), and for specifying constraints on the position of ports. It is related to APL and a working knowledge of APL is helpful in writing Earl programs. Like APL, it includes features for manipulation of lists, changing their elements, arithmetic, etc. Functions have a syntax similar to APL, but Earl has a more familiar precedence. On the other hand, Earl is not, nor is it meant to be, a complete general purpose programming language. Instead, Earl is specialized to the task of designing primitive cells and connecting them together.

At Caltech, the most commonly used design tool up to now, LAP, was based on fixed cells. The designer had to keep track of where the ports of each cell were, through each coordinate transformation, as cells were drawn. Earl takes care of that detail, assuring the designer that wires line up correctly. Though the layout primitives spring directly from CIF[1] and LAP, the discipline of describing the stretchable layout actually makes digitizing the layout easier, and changing it becomes vastly easier.

Earl was written in the programming language C, and runs under Berkeley UNIX[2] It is being used by the integrated circuit design class at Caltech.

1.1. Overview

In Earl, the collection of variables, functions and cells defined is called a workspace. A workspace can be saved in a text form that can be edited and read back by Earl. A workspace will usually contain only one chip.

A chip is a collection of cells, along with any necessary functions. Cells are declared at the same lexical level – they may not be nested inside one another. A cell is a rectangular area on a chip that has interface locations, called ports, and programs to execute that describe its layout. A port is a point whose coordinates will be computed by the constraint solution mechanism of Earl. The coordinates of the points are given to the cell to be used in the generation of its layout.

A cell declaration has three parts: port declarations, constraint statements, and the layout of the cell. Cells can be parameterized so that similar cells can be made with one specification. Pure composition cells have little more than a list of the cells to be composed. (This is where the user of Earl wins over other systems.)

[1] Described in [Mead&Conway 1980] chapter 4

[2] UNIX is a Footnote of Bell Laboratories.

Ports are declared either in groups, or singly. A group is an ordered list of ports used for connecting cells together. Singly declared ports are internal to the cell, and serve for making constraints and layout. Four of the groups, north, east, south and west, are special and serve as the edges of the cell for connection. All cells have these groups, even if they are empty.

Constraints relate x (horizontal) or y (vertical) coordinates of two points. Constraints specify either a minimum on some directed distance, the minimum value of one coordinate minus another, or the exact distance. Constraints are used to build a graph where the nodes are collections of points that are constrained to move together, and the arcs are the stretchable constraints.

The layout is made of wires, boxes and contact cuts. Coordinates used in the layout are all relative to some point. The basic philosophy of the system (a subset of "Boston" layout) is that lines at any angle and circular arcs are allowed.

Functions are used as infix operators. User defined functions become infix operators in the language.

1.2. A Simple Leaf Cell

The following is an example that shows how these concepts are used in Earl designs. All leaf cells can be designed following this pattern.

This cell makes a poly box, and a metal wire that comes 'in' on the north and goes 'out' the east.

```
cmos;

cell corner;           /* simple wiring cell */
north group in;         /* "in" is on the north */
east  group out         /* and "out" on the east */

constr
xcon west |>3| in |>4| east;       /* constrain x coordinates */
ycon south |=2| out |>6| north      /* constrain y coordinates */

geom
metal wire in,in.x#out.y,out;       /* make a metal wire */
poly  box west.x+1#south.y+1,east.x-1#north.y-1    /* and a polysilicion box */
end
```

The first line

```
cmos;
```

tells Earl that this design will be in cmos. The next line

```
cell corner;          /* simple wiring cell */
```

begins the definition of a cell, named 'corner'. Characters between /* and */ are comments, and comments nest. That is, /* /* *this* is a comment */ and so is this */ but this is not.

The lines

```
north group in;        /* in is on the north */
east  group out;       /* and out on the east */
```

create a port called 'in' in the north group, and 'out' in the east group.

After ports have been created, you must constrain their positions in x and y. The keyword **constr** begins the section of the cell where the constraints are made using the functions **xcon** and **ycon**. Their argument is ports or groups alternating with distance specifications. The distance specification between two ports constrains the distance between them. The lines

```
constr
xcon west |>3| in |>3| east;        /* constrain x coordinates */
ycon south |>3| out |>3| north;     /* constrain y coordinates */
```

begin the declaration of constraints and then make some constraints in x then y. The constraint specification |>3| means that the distance from 'west' to 'in' is greater than or equal to 3. Remember that the distance is directed! "xcon west |>3| in" means something completely different from "xcon in |>3| west." **Nothing** more than the stated constraints are implied. If east is more than six more than west, you are not guaranteed to have 'in' in the middle of the cell. (In fact, there is no way to constrain anything to be exactly in the middle of anything at all.)

The constraints specify the range of legal coordinates for a point. Earl computes how the cell must be stretched to fit its environment, and executes the statements following the keyword **geom** for each different stretching. In that section the ports are used as coordinates to make the layout.

The statements

```
geom
metal wire in,in.x#out.y,out;        /* make a metal wire */
poly  box west.x+1#south.y+1,east.x-1#north.y-1;    /* and a polysilicion box */
```

make a single metal wire that runs from the port named 'in' to the port named 'out' making a right angle bend, and put a box in the cell. The optional left argument to **wire** can have a layer name, telling what layer to put that wire on, and a number telling its width. (The layers available depend on the technology; see the description of the layout section of a cell for details.) If you don't specify a layer, it uses the last layer used. If a width isn't specified, it uses the minimum width of the layer. The right argument is a list of coordinates describing the center line of the wire. The second element of the list is a computed coordinate. In Earl, all numbers and coordinates are complex numbers. The # operator constructs a complex number with the given real and imaginary part.

Box has an optional left argument, which is the layer to put the box on. The layer defaults to the last layer used. The right argument must be a list with exactly two values, defining opposite corners of the desired box.

When you type this cell to Earl, it responds "--corner" telling you that it has defined the cell corner.

When you have a cell defined you can produce a CIF file and route it to various destinations. The function **make** will write a CIF file on the disk, **plot** will send CIF to cifplot to be rasterized in black and white, **colorplot** will do the same for color, and **cifp** will send CIF to a pen plotter (like an HP7221, or a Charles terminal). For example:

```
plot corner;
```

1.3. Composition — horiz and vert

The easiest way to connect cells is to make a horizontal or vertical composition. Functions called **horiz** and **vert** take a name for the resulting cell on the left, and a list of 'instances' on the right. An instance is the combination of a cell definition with a transformation. Functions to transform instances are **mx** *instance* (negate the x coordinate), **my** *instance* (negate the y coordinate) and *instance* **rot** *number* (rotate the given instance some number of times 90 degrees counterclockwise). For example,

```
row horiz corner,mx corner;
```

will make a cell 'row' that is the horizontal composition, from west to east, of a corner and a corner mirrored in x. If there is no name for the resulting composition it is called 'horizcell' or 'vertcell' and an instance of it is returned. This can be used in a statement such as

    plot horiz corner, mx corner;

which will create, plot, and then delete this composition cell. The cells are connected in order of increasing coordinate; corner will be to the left of corner mirrored in x. vert is basically the same, but it stacks instances up vertically, south to north. The statement

    diamond vert row, my row;

will take a row and connect it to a row mirrored in y.

2. Interpreter Language

2.1. Data

2.1.1. Null

Null is entered as **()**.

2.1.2. Identifiers and Scope

Identifiers are made up of letters, digits and underscores; a letter or an underscore must be first. Note that upper and lower case are distinct. Only the first twelve characters are significant; all extra characters are quietly ignored. Keywords (e.g. if, constr, cell) and standard function names (e.g. cos, group, print) are all reserved and may not be used for your variables.

Variables do not need to be declared before use; the type of a variable depends on what was last assigned to it. In other words: data has types, variables do not.

When Earl evaluates an identifier, it looks in the current symbol table. If the identifier is not found, it is created. Associated with every identifier is a flag that tells if its value is global, or local to the current cell or function. Global identifiers are those referenced outside of a cell or function, and can be made accessible in a cell or function only with an **external** declaration.

2.1.3. Numbers

Numbers in Earl are represented by complex numbers. A single number serves as a coordinate pair for doing layout, where the real part is x, and the imaginary part is y. Each component of a number can be written in the usual way, except that exponential notation is not allowed. (where in a circuit design would you really need to have 4.3e19?) The components of the complex number are separated with #, and arithmetic handles complex numbers properly.

Numbers have three components which can be accessed with the "." operator. They are: ".x," the x coordinate; ".y," the y coordinate; and ".r," the magnitude.

Legal numbers are: .5, 3, 2#1

2.1.4. Strings

Strings are enclosed in double quotes (") and may not include a newline unless it is preceded with a backslash (\). Other meta-expressions available are: **\t**, tab; \\, backslash itself; **\b**, backspace; **\r**, carriage return; **\n**, newline; **\f**, form feed; \", a quote; \nnn, the character represented by the octal number *nnn*. When strings are the result of an expression or are printed with the **print** function they appear without the quotes and all the meta-expressions are expanded.

ex: "Hello, World\n"

2.1.5. Lvals

An lval is the structure that represents the location of a value. It is produced when an identifier is evaluated. The existence of lvals brings about many useful effects in the language which will be pointed out later.

2.1.6. Lists

A list is a variable size collection of objects. A comma separates elements, except that when a list is printed no comma is put on either side of a string. When a list is created by concatenating elements together, nulls are lost; however, null can be put into a list by assigning to a specific element. A list can be an element of another list; braces ({}) enclose a list that is an element of a list. An lval can be an element of a list, which means that with a single assignment statement many variables can be changed concurrently. However, a list within a list, and a list put into a variable cannot contain lvals. Here are some examples of what can be done with lists:

```
(1,2,3) , (4,5,6)       ☛1,2,3,4,5,6
{1,2,3} , () , {4,5,6}  ☛{1,2,3},{4,5,6}
{1,2,3} , (4,5,6)       ☛{1,2,3},4,5,6
a:={1,2};               ☛{1,2}
b:={3,"Hi"};            ☛{3,Hi}
a,b;                    ☛{1,2},{3,Hi}
{a,b};                  ☛{{1,2},{3,Hi}}
a,b:=b,a;               ☛{3,Hi},{1,2}   /* correctly exchange values */
{a,b}:={1,2}            will not work
```

2.2. Operators

2.2.1. Arithmetic Operators

Assignment is :=.

The binary arithmetic operators are +, #, -, *, /, % (modulus), and ^ (exponentiation). The ordinary ones do what you would expect on numbers. # takes the value on the right, multiplies it by *i*, and adds it to the value on the left. It has a precedence lower than all the other operators, so 3-1#5 is 2#5 (not 2#-1). It can also be used as a unary operator; ##3 is -3. Exponentiation doesn't work for complex numbers yet.

has the lowest precedence, - and + come next, *, /, and % after that and ^ is the highest. All but ^ group left to right. For example:

```
2+3           ☛5
5*2+3         ☛13
(4#-3)*(4#3)  ☛25
4#-3*4#3      ☛4#-9
```

Any one of these operators may be followed with '=' to make it assign the result to the left hand argument. var+=3, has a value of three more than var was, and changes var to that new value. Used this way, they all have the same precedence, that of assignment.

2.2.2. Logic Operators

Equality is =, and not equals is <>. They have the same precedence. Above them is <, <=, >=, and >, all with the same precedence. Their precedence is lower than the arithmetic operators, as would be expected. The result of each of these is either zero, or one. Any nonzero number is considered true, and zero is false. Relational operators can operate on lists, just like arithmetic operators.

3 < (1 to 5)☛0,0,0,1,1

Logical functions are **and**, **or** and **not**. These work on boolean values, not bits. There are no functions to do any bitwise operations. **And** and **or** always evaluate both arguments. If you don't want both arguments to be evaluated, there is **andif** and **orif**. They are equivalent to **and** and **or** except that if the first expression determines the value, the second one is not evaluated. (Also, there are synonyms **&&** and **||**.)

2.2.3. List Operators

The symbol @ is the value of the previous element of a list or the previous expression. This is useful in describing the layout of a cell. For example: 3,@#3,3+@☛3,3#3,6#3.

Lists also can be created with the function **to**. *a* **to** *b* gives a list of numbers from a to b, with a difference of one between the elements.

```
1 to 4   ☛1,2,3,4
4 to 1   ☛4,3,2,1
```

List difference is the function **without**, which takes each element of the right argument, and removes the first occurrence of it from the left argument. It is okay for an element of the right not to be found on the left.

```
1 to 5 without 3             ☛1,2,4,5
1 to 5 without 3,6           ☛1,2,4,5
1 to 5, 1 to 5 without 3     ☛1,2,4,5,1,2,3,4,5
```

The function *thing* **rep** *count*, makes a list of count copies of arg. This is most useful in composing cells. I recommend that instead of listing a cell to be composed many times, that **rep** be used so that the design can be easily parameterized.

```
(1,3) rep 2          ☛1,3,1,3
regcell rep nregs    ☛a list of regcells
```

Arithmetic also works with lists. Two lists must have the same shape to be able to operate on them. Two things are the same shape if: they have the same length and corresponding elements are the same shape; or one is a single thing and the other is a list. If you operate on a list and a number, the result is a list of the same shape as the list, where each element has been operated on with the number. Thus:

```
a:=1 to 3,{4 to 6};   ☛1,2,3,{4,5,6}
a+1                   ☛2,3,4,{5,6,7}
a*2                   ☛2,4,6,{8,10,12}
a#a                   ☛1#1,2#2,3#3,{4#4,5#5,6#6}
```

Square brackets ([]) are used to index into lists, like arrays in most languages. 1 selects the first element of a list. Indexing creates an lval for the element. If a value gets assigned to it, the element need not have been in the list before; the list will be extended as needed. If the value is used, then, of course, it must have a value. The index expression can itself be a list; in which case the result is lvals referencing elements of the original list in the order and structure of the index expression. For example, using the above value of a;

```
a[1]             ☛1
a[{1}]           ☛{1}
a[4]             ☛4,5,6
a[1,3,{2,3,4}]   ☛1,3,{2,3,{4,5,6}}
a[2,3]:=a[3,2]   ☛3,2 and 'a' now is 1,3,2,{4,5,6}
```

If an element of the list is itself a list, you must subscript again to get the elements of the sublist. a[4][2]☛5.

The function **len** *list* returns either the number of elements in a list, 1 if it is not a list, or 0 if it is null.

2.2.4. String Operators

Strings can be compared with any of the relational operators. Strings are concatenated with +. String concatenation is recursive. That is, concatenating a string to a list of strings will produce a list of string where each element is the concatenation of the string and the corresponding element from the argument list. Relational operators work on strings, just like they do on numbers. **max** and **min** work on strings, and return the larger or smaller argument. (for clarity, the results of the examples are written with quotes and commas)

```
"Best of all" + " possible worlds"  ☛"Best of all possible worlds"
"reg" + ("0","1","2","3")           ☛"reg0","reg1","reg2","reg3"
"a"="A"                             ☛0
"a"<>"A"                            ☛1
"summer" min "adder"                ☛"adder"
```

2.2.5. Precedence Summary

In the list below, operators on the top lines of the list below will be evaluated before operators on the lower lines. For operators of equal precedence, evaluation proceeds from left to right except for the assignment operators.

```
^  (right to left)
* / %
+ -
#
< <= > >=
= <>
not
and
andif  &&
or
orif   ||
to  rep rot mx  my, functions that make elements of lists
,
all other functions, including user written functions
:=  +=  *= etc.     (right to left)
```

2.3. Control Flow

Statements in Earl are separated with a semicolon. Unlike most languages, there is no such thing as a begin-end pair. Instead each control structure has its own terminating keyword which is always required.

2.3.1. If Expressions

An if expression executes one set of statements depending on a control expression. Using a modified BNF (**literals**, {zero or more of these}, [zero or one of these]), the syntax is:

> *ifExp* ::= **if** *exp* **then** *actionStats*
> {*efStat*} [*elseStat*] **fi**
> *efStat* ::= **ef** *exp* **then** *actionStats*
> *elseStat* ::= **else** *actionStats*
> *actionStats* ::= *actionStat* { ; *actionStat* }

The *exp* must produce a number. An *actionStat* is any statement except a definition of a cell or function. **ef** means 'else if' and is just a convenience to save from writing a pile of fi's at the end.

The result of an if is the result of the last statement evaluated. If none is selected, then the result is null.

2.3.2. Iterative Constructs

The iterative constructs are the **while**, **repeat-until**, and **for** loop. They are written:

> *whileStat* ::= **while** *exp* **do** *actionStats* **od**
> *repeatStat*::= **repeat** *actionStats* **until** *exp*
> *forStat* ::= **for** *name* := *exp* **do** *actionStats* **od**

The while and repeat-until do the usual thing. The *exp* in *forStat* should produce a list. Each value of the list is assigned to the variable and then the *actionStats* are evaluated. *name* must be a simple variable; no indexing or anything else. There is no break or continue to alter the execution of the loops.

2.3.3. Other Control Structures

There are no others. No goto. No case statement.

3. Cell Declarations

This is where we get down to the nitty-gritty. The syntax of a cell declaration is:

> *cellDef* ::= **cell** *name* [*cellParams*] **;**
> *cellGuts* **end**
> *cellParams*::= **(** *name* { **,** *name* } **)**
> *cellGuts* ::= *actionStats*
> [**constr** *actionStats*]
> [**geom** *actionStats*]

The name following the keyword **cell** is the name the cell will have. The optional *cellParams* define parameters for the cell. When a cell with parameters is used, a value must be given for each parameter. If there is only one parameter, a list can be given without braces. If there is more than one parameter, then the values given must be in a list of the right length. A list could be given to any parameter, if the list is in braces. Typical uses for cell parameters would be telling a buffer to invert or not, telling a pad how strongly it should drive, etc. For example, the declaration,

```
cell SBuffer(non);
   (body of cell)
end
```

defines a parameterizable cell called SBuffer, with a single parameter called non. The only thing that can be done with a parameterizable cell, is to give it parameters to make a "fixed" cell. For example,

```
bufpair horiz SBuffer(0), mx SBuffer(1);
```

makes bufpair an inverting superbuffer connected to a non-inverting superbuffer that is mirrored in x.

When a cell's symbol table is created, the layer names and lambda[3] are automatically declared external. The syntax to declare other external symbols is

externalDecl ::= **external** *name* { **,** *name* }

3.1. Port Declarations

The first section of code for the cell creates the ports and groups which will be used in the stretching of the cell. Groups of ports are created to carry information about ports up the hierarchy. The groups named east, north, west and south contain the ports which are connection points to the cell. They are special in that transforming the cell changes their contents, and all the ports in them are made externally accessible. All groups are composed by ordinary composition functions. That is, if a cell has a group called power, all the composition cells that use it will have a group called power containing all the external ports of its subcells that were in the power group. Ports not in a group may be used in a cell to use the composition algorithm to figure out where to place the layout. Groups and ports are declared with the group and points functions.

name **group** *listOfNames* ;
  points *listOfNames* ;

The first name will be the name of the group, the names following are the names of the ports in the group. The ports in east, north, west and south **must** be in order of increasing coordinate. If they are not, when you go to compose the cell, it won't fit and you'll get a terrible, cryptic error message. There is hope though. The function **check** *instance* checks a cell, or a given parameterization of a parameterizable cell, to see if you have the ports out of order.

Ports have components x, y and c. x and y are just the value of that coordinate. c is used to turn a port into a number. North and south have a component y; east and west have a component x. They can be used in the layout to represent the coordinate of the edge of the cell. e.g., "north.y;"

A port can be in more than one group, but not on adjacent edges of the cell. (North and south are okay, but not north and east.) When you use the coordinates of a port on two edges, you get a list of two coordinates, sorted in order of increasing coordinate. That is, if vdd is on the east and on the west, vdd acts like a list of two numbers. The first one is for the west, the second is for the east (east.x is larger than west.x, right?) If you access the x or y component of such a port, it will either give you a single number (vdd.y is a single number, the height of vdd on both sides) or a pair (vdd.x is the x coordinate of vdd on the west, followed by the x coordinate of vdd on the east).

The ports can depend on the parameters in several ways. A statement like

```
west group ground,if doubleBus then bus2 fi, bus1, vdd;
```

will include the port bus2 only if doubleBus is non-zero. (if it is zero, the if-expression will return null. When null is concatenated with something, it is lost.) Or, another useful thing to do, is to make the ports in a list.

```
south group gate[1 to 2*nInputs];
```

will make a two ports for each input, named gate[1], gate[2], and so on. (gate[1 to 2*nInputs] makes a list of 2*nInputs lvals. group assigns a port to each lval and gives the port the name of the lval.)

[3] [Mead&Conway 1980], section 2.6

3.2. Constraint Section

This section is preceded with the keyword **constr**. Here the user specifies constraints on the coordinates the system may assign to the ports so that the layout will satisfy design rules. Constraints deal with the directed distance from one port to another, either specifying the exact value, or its minimum. Associated with each port is a node of the x and a node of the y constraint graph, and a separation from them. Specifying an exact distance between two ports causes them to reference the same node, with possibly different offsets from it. Specifying a minimum value adds an arc to the constraint graph with a minimum value for the difference in the coordinates.

Constraints are specified with the special functions **xcon** and **ycon**. The syntax of the argument to these functions is:

constrList ::= *portOrGroup*
  | *constrList* "|" *relation* [*exp*] *portOrGroup*
relation ::= < | <= | = | >= | >

That is, a port or a group, followed by repeated groups of a bar, a relation, an optional expression, another bar and a port or a group. ('>' is synonymous with '>=', as is '<' to '<='). It means, make a constraint between the ports or groups such that the distance from the first port to the second port will fulfill the relation to the value of the expression. The value is zero if the expression is omitted. This is applied to each pair along the list. For example:

```
xcon west |=| phi |>5| load;
```

first will make the x coordinate of phi be equal to the x coordinate of the west edge of the cell (and all the ports on it), then will constrain the distance from phi to load to be greater than or equal to 5.

Constraints can be in one dimension only; you cannot constrain something to be square, for instance. A constraint can depend only on the distance from one port to another; you cannot constrain something to be 'in the middle' of two other ports. If you specify constraints that cannot be satisfied, (for example, "xcon load |>5| read |>5| load;") you will get an error that will be cryptic and hard to interpret. I promise. (cross my heart and hope you won't ask me why[4])

3.3. Layout

Earl passes control to this section once for each different way a cell gets stretched. This is vital to understanding how Earl works. First it solves the constraint graphs, then it actually produces the layout based on it.

The layout follows the keyword **geom**. Here the user takes the given coordinates and makes wires and boxes using them. Stretching is accomplished by merely giving different coordinates to the ports. The layout is specified with the functions described below.

3.3.1. Layers

There is no function to set the current layer, just optional arguments to the layout functions. The names of the layers are:

[4] but you can read appendix B

nMOS	cMOS/SOS
diff	island
implant	implant
poly	poly
metal	metal
glass	glass
cut	cut
buried	

3.3.2. Contact Cut Structures

Depending on the technology specified, there are various structures available for connecting the various layers together. Their names are constructed from an abbreviation for the layers they connect together. The asymmetric contacts also indicate the orientation. Each contact is a function that puts the specified contact at each coordinate in its argument list and returns its argument so it can be used in statements.

3.3.2.1. cMOS/SOS Contacts

([abc] means one character from the set a, b, c)

im	island to metal
pm	poly to metal
ii[en]	island to island east/north
ip[enws]	island to poly
ipi[en]	island to poly to island

iin is used to short a diode and has its long axis pointing north/south. ipe is a butting contact from island to poly, where going from island to poly is going east. ipie is an 8 by 4 lambda structure to short a diode and contact to poly at the same time.

3.3.2.2. nMOS Contacts

dm	diffusion to metal
pm	poly to metal
dp	diffusion to poly buried contact
dp[enws]	diffusion to poly butting contact

(The designer should know what he can get fabricated before using butting or buried contacts.) These follow the same conventions as the cMOS/SOS contacts: dpe connects diffusion to poly, pointing east, centered on the given coordinate.

3.3.3. Boxes

Earl only has boxes Manhattan style; boxes are oriented with their edges running only horizontally and vertically. A box is specified with an optional layer, and a list with exactly two coordinates which are the coordinates of opposite corners. For example, a unit poly box: "poly box 0#0,1#1;"

3.3.4. Wires and Polygons

Wires have a width, layer and path. If not specified, the layer defaults to the last layer used, and the width defaults to the minimum width for the current layer. Wires extend half the **minimum** width of the current layer beyond the end points of the path. Polygons have a layer and a path. The default layer is the previously specified layer.

A path cannot always be completely specified with a list so there is a current path. The **wire** and **polygon** functions set the current path, and return it. The functions **miss** and **seg** add arcs and lines to the current path.

A path returned from the wire or polygon function can used as an argument to wire or polygon later on. This lets the designer work on several paths bit by bit, instead of having to finish one before beginning the next. The left argument to wire is optional and can be a path (previously returned from wire), a layer, a width, or a list of layer,width. The right argument is either a path or a list of points. If either of the arguments is a path, the current path is set to it and more segments can be added on the end. If neither argument is a path, a new wire is made on the given layer of the given width (with the appropriate defaults). If points are given on the right, they are points on the path of the wire. The polygon function works exactly the same way, except that you can't specify a width. For example:

```
  metal,4 wire busin,@+2;
 gndrad*2 wire gnd;
    poly  wire busin+3,east.x-1#bus.y,bus;
       4  wire gndsmall,east.x-.5#@.y;
    poly  polygon a,b,c;
```

The first wire sets the current layer to metal, and draws a 4 wide wire that starts at the port called busin and goes over 2 in x. The second wire computes its width, and draws a wire from gnd on one side of the cell to gnd on the other (remember if a port is on two sides, using it in arithmetic gives a list of two coordinates.) The third is on poly with three points on the path. The last is 4 wide on poly. The poly polygon is a triangle.

The function **miss** *point,radius* appends a potential arc to the current path. If the radius is positive the path will wrap counter-clockwise around the point; if negative, clockwise. The path will actually only wrap if it needs to bend to miss the point. If going straight will stay far enough away from the point, the miss is ignored. The coordinates of the path are computed to make tangents based on the previous and following points and arcs; the user does not need to find where on the arc the path should hit. If there is no tangent (a tangent to a circle cannot contain any point in the circle), you will get the message "circle in a circle;" and marks will be put on the comment layer to help you locate the problem. (The arc is actually approximated with a polygon. Currently 12 sides are used; that many appear to be quite sufficient. All the polygons bend at the same angle, so design rules are not violated.)

Neither the first nor the last parts of a path may be a miss — sometime after a miss you must put a **seg** on the path. The function **seg** *list of points* adds more points to the current path. It is only needed after an arc; other times it is merely redundant.

```
poly wire e-2; miss e+1,-3; miss q,4; seg out;
poly wire c+2#-7; miss f,-4;
    miss e+1,4; seg c+4#6;
diff wire a.x+4#g.y-19; miss b+3,3; miss b+1#2,3;
    miss (pm c+2#-6),-5; seg c-3#-7;
metal wire pm a,b;
metal polygon a#-2; miss a,-2; seg a#2; miss a+2,-2; seg a#-2;
```

Notice in the third wire that a coordinate being missed is the result of a contact cut function. The fourth wire puts down two contact cuts and runs a metal wire between them. Using statements like that, related layout features can be kept together in the source.

3.3.5. Electrical Nodes

To help support Tom Hedges' circuit extractor, there is a function to output electrical node names into the CIF. The function is:

[*layer*] **elecnode** *location* [,"*node name*" [,"*node type*"]]

The optional layer determines which layer is meant at the given location. The node name is a text string for the desired name. The node type is a text string whose meaning is defined by the extract program. If the location is a variable, the node name defaults to the name of the variable. (This function causes the information to be put into the CIF with a user extension. The extension is 84 x-coord y-coord node-name opt-node-type.)

3.3.6. Including CIF

Finally, there is a function **drawcif** so that you can include cif produced by another design tool. All it does is copy the cif, it does no checking, no stretching. It's so dumb it doesn't even read connectors to the cell so that you can connect to its "ports." The only thing that it does do, is to include other cells referenced by the desired one, until it can fabricate the requested cell. **drawcif** is used like this:

["*file*"] **drawcif** "*cellnam*",*mx*,*rot*,*location*;

The "*file*" is an optional file to read the stuff out of. If omitted it is ~earl/lib/nmos or ~earl/lib/cmos. the "*cellnam*" tells what cell to get. *mx* is nonzero if you want it to be mirrored in x, *rot* gives how many times to rotate it by 90, and *location* is the coordinate of where the cell's origin will go inside the Earl cell drawing it.

4. Composition Cells

Cells are composed by abutment, with stretching to resolve mismatch. It is up to the designer to discover if routing would save area. If so, it's also up to him to do it, making a cell that does the necessary work. (It isn't very hard anyway, since Earl will stretch the routing cell to fit too.)

4.1. Transformations

Because of the restrictions of the stretching algorithm, only mirroring, and rotation by multiples of 90 degrees can be used. The functions are:

mx *inst*	negate the x coordinates
my *inst*	negate the y coordinates
inst **rot** *num*	rotate *inst*, *num* times 90 degrees counter clockwise.

The final position of a cell is computed from the constraint graph; there is no translation.

4.2. Naming Instances

For the circuit extractor, instances of a cell can be named to be able to distinguish one electrical node from another. The function is

insts **named** *names*

There must be as many instances as names, or just one instance or one name. The function returns a list of instances that can then be composed. (This information is put into the CIF file with a user extension. The extension is 85 instName; The name applies to each following call until the next 85, which can have no name at all.)

4.3. Composing

All that is necessary to specify for a composition cell is a list of instances to be composed, and the direction to stack them. Optionally you may give the composition a name to save it for later. If you don't give it a name, an instance of it is produced and must be used right away.

Composition works by constraining corresponding ports on the edge between the cells to be at the same place. (This is why the order of ports matters.)

All this magic is done with the functions **horiz** and **vert**. They take an optional left argument which can be a name, a merge parameter, or a list of a name and a merge parameter. On the right is a list of instances that are to be composed in this manner. The resulting cell is again constrained to have rectangular edges.

Sometimes two cells are to be connected so that they share a wire. If each cell has a port for that wire, the composition would have two ports at the same place. That is likely to cause trouble when another cell tries to connect one wire to one of those ports, and another to the other. For this there is an option to **horiz** and **vert** called merge. Merging in a horizontal composition (in addition to joining corresponding ports on the east of the first cell to those on the west of the second) joins the ports on the north adjacent to the connection, producing one port on the north of the composition, and joins the ports on the south adjacent to the connection. Merging joins ports together, but doesn't affect the layout of the cells in any special way. You can 'merge' on both sides, or just on one with the words [enws]merge. Ports that are to be merged **must** be allowed to be on the corner of the cell. If not, the constraint graph will be broken and you will get cryptic messages.

```
bufset horiz (SBuffer(1), mx SBuffer(1)) rep 2,SBuffer(1);
groundend vert botrowgnd,my groundcell,(gndpair rep nbits/2-1);
regpair,merge horiz mx regcell,regcell;
gndpair,wmerge vert groundcell,my groundcell;
```

The first example makes a row of five SBuffer cells. It repeats a list of an SBuffer and a mirrored SBuffer twice, producing a list of four where alternating ones are mirrored. Next it concatenates one more SBuffer onto the end. Horiz composes these together and calls the cell produced bufset.

The second example makes a column of botrowgnd, groundcell mirrored in y, then some number of gndpair cells.

regcell has a clock wire running up the west edge which is to be shared by pairs of cells. The third example takes two of them, and connects the ports between them, and the clock ports on the north and south. (A single regcell has 3 ports on the north. This composition has only 5.)

On the west edge of grouncell there is a wire that is to be shared.

When two cells must be connected with some ports left out, extra, "dummy", ports can be added to one of the cells. The extra ports connect to the ports that are to be left out, and then ignored in the layout of the cell.

This method is not the most general way to compose, since it cannot produce some structures. But it is general enough, and so easy to use, that the methods outlined in the next sections need to be used only rarely.

4.4. More General Composition

Horiz and vert evaluate the instance list in the context of a new cell, not in the context where they are written. If you want to compute the list and not use global variables, declare a cell, compute the list of instances, and call **hcompose** *list of instances* or **vcompose** *list of instances.*

These work just like horiz and vert, but they are normal functions, their argument gets evaluated in the current context. They take an optional left argument which is the merging parameter, just like horiz and vert. You may call them only once in a cell because it turns the cell into the composition.

```
cell PlaPair(l,r);
external placell,merge;
merge hcompose placell(l), mx placell(r);
end

cell PlaRow(r);
external placell,PlaPair;
for i:=(1 to floor (len r)/2)*2-1 do
   ls[i]:=PlaPair(r[i],r[i+1])
od;
if (len r)%2 then ls[len r]:=placell(r[len r]) fi;
hcompose ls;
end
```

PlaPair makes a merged pair of placells; PlaRow makes a row of PlaPairs with an extra placell on the end if necessary. Notice that the cells and merging parameters must be declared to be external to the cell.

4.5. Messy Composition Cells

(This section shows the detail function of composition. It may be skipped until you have more familiarity with Earl.) Complicated composition cells use the first section of a declared cell to draw other cells, connect them up, and make their groups from the groups of the subcells.

In order to allow greater flexibility, instances of cells can be transformed as desired. When the transformation is complete, the function **draw** *instance* instructs the system to draw the instance when this cell is drawn. Draw returns an instance which is unique to that drawing of it. (That is, for "a:=b," a and b point to the same instance. For "a:=draw"b, they point to different ones. The one a points to will be drawn.) Another thing it does, is add the transformed constraints of the instance and its ports to the constraint graph of the cell.

Once an instance has been drawn, its groups and external ports can be accessed. They are accessed by instance . name of the group or port desired. That is, "a.north" is a list of ports that correspond to the north edge of the instance in variable a. (It is the north edge of the instance, not of the cell that it is an instance of. If the instance is rotated or mirrored, the ports and/or the order of them in the list may be changed.) These ports can be either connected to another cell, or made external to the composition cell.

The function *list of ports* **connect** *list of ports*, takes pairs of ports, one from the first list with one from the second list, and connects them together. It does this by constraining the ports to be at the same place in the final layout.

An example of when one might want to do such a terrible thing, is if not all the ports from a cell can be connected to another, for instance, a register array with ports on the east and west that are vdd, bus, gnd, bus, etc., and it is to be connected to a cell that gives it power on the west, and ground on the east. Here is the code that would do that.

```
    cell regblock;
    external emerge, array, groundend, prechpair, nbits;
    prech:=draw (emerge vert (prechpair rep nbits/2));
    arr:= draw array;
    gend:= draw groundend;
    north := arr.north, gend.north;
    south := prech.south[2], arr.south;
    west  := prech.west;
    east  := gend.east;
    prech.east connect (arr.west without arr.ground);
    (arr.east without arr.power) connect gend.west;
    end
```

(This is what horiz, vert, hcompose, and vcompose do)

5. Built In Functions

5.1. Output CIF

Once a design is specified, You have to output it in a format that other programs know how to read. The best known format is CIF, and in fact that's all you get. For convenience, Earl provides a variety of destinations.

5.1.1. CIF to a Disk File

The function 'make' takes an optional left argument for the file name to write the CIF to, otherwise it writes to *cellName*.cif On the right must be an instance of a cell.

5.1.2. Piped to cifp

The function 'cifp' opens a pipe to the cifp program and feeds it a CIF file. The right argument is an instance of the cell to draw. The optional left argument is used as a command line argument to the program. This can be used to change what kind of plotter to drive, and set the physical device that it will send the plot commands to. The command line options that cifp knows about are:

-p*plotterType*
: Define the kind of plotter to drive. *plotterType* is an abbreviation for the plotter that is hooked up to the device selected. The plotters we have around are called Charles, HP7221, GIGI, and 8COLOR. (It can also drive some others, but they are not commonly connected.) The default is GIGI.

-d*deviceName*
: Send the plotter commands to the file specified by *deviceName* (no space between the '-d' and the device name). The default depends on the plotter selected.

As soon as the CIF has been read, cifp will begin to plot. For complete information on cifp, read the manual page on cifp. (it is a different program that Earl knows how to deal with. It is not a part of Earl.)

5.1.3. Piped to cifplot

The functions **plot** *instance*, **colorplot** *instance*, and **colourplot** *instance* open a pipe to the cifplot program and pipe it CIF. The right argument is an instance of the cell to draw. Earl figures out whether to add the commands -c (color), and -x (cmos). Other arguments can be provided by modifying the variable 'PLOTARGS'. It just has text which is appended to the command line for cifplot. For the meanings of the arguments, read the manual page on cifplot (it is a different program that Earl knows how to deal with. It is not a part of Earl.)

5.2. More Math

There is **sin** and **cos** which work correctly on complex numbers.

5.3. Input and Output

Earl keeps one file for input and one for output for programs being interpreted. They start out being the standard input and output files. They can be re-directed with the functions **infile** *fileName*, **outfile** *fileName*, and **appfile** *fileName*. The *fileName1 is a string which can use the ~loginName* to represent someone's home directory. Infile opens the input file; outfile and appfile open the output file; outfile overwrites the file and appfile appends to it. These functions return 1 if they succeed, and 0 if they don't.

Anything evaluated from the terminal is automatically typed back out. If you wish to get a function to type something out, **print** *exp* will print the expression into the outputfile. It does not do too much to the arguments, that's up to you. In particular you must put a newline ("\n") where you want one.

The function **getchar** returns a number corresponding to the ascii value of the next character available on the input file, or -1 at the end of the file. **getstring** returns a string containing the next line of text available from the file, including the newline character. It returns null at the end of the file. **geteval** parses an expression from the input file, evaluates it, and returns the result. Any symbols in the expression are looked up in the global symbol table. Null is returned at the end of the file, but since many expressions evaluate to null (like a for statement), it is not possible to tell when you are at the end of the file.

5.4. String Functions

strlen *exp* gives the length of a string, or -1 if the expression is not a string. *string* **substr** *exp* returns a string where each character is selected from the argument string based on the value of the corresponding element of the expression. The function **cvs** *exp* will convert numbers and lists of numbers to strings or lists of strings.

For more convenient treatment of strings, there is ascii <exp>. It turns a number into a string with one character and a string into a list of numbers. If it gets a list, it converts consecutive numbers into a string with those characters, strings into consecutive numbers, and sublists are treated recursively. (ascii is almost its own inverse. It breaks down if a list has consecutive strings, or the numbers do not exactly match characters.) (for clarity, the results of the examples are written with quotes and commas)

```
strlen "foo"           ☛3
strlen ""              ☛0
strlen 3               ☛-1
"Proteus" substr 3     ☛"o"
"Earl" substr 5,4,3    ☛"lr"
cvs 1                  ☛"1"
cvs 1 to 3             ☛"1","2","3"
cvs 1,{2,3}            ☛"1",{"2","3"}
ascii 32               ☛" "
ascii 48               ☛"0"
ascii "foo"            ☛102, 111, 111
ascii ascii "hello"    ☛"hello"
ascii 32,"foo"         ☛" ",102, 111, 111
ascii "A",{"foo"},32   ☛65, {102, 111, 111}, " "
```

5.5. Errors

If a program detects an error and wants to stop processing, **abort** *exp*, will print the expression on the terminal and abort the execution of the program.

5.6. Miscelaneous Functions

The function *item* **index** *list* returns the index of the first place in the list where the item is found, or 0 if it is not found. If the item is a list, index recurses on itself, returning a list. This can also be used to determine if something is in a list. If it is in the list, the result will be non-zero, which is true, otherwise the result will be zero, which is false. It must be legal to compare the item to each element of the list or index may fail.

4 index 3,6,4,2	☛3
5 index 3,6,4,2	☛0
4,5,{6,3} index 3,6,4,2	☛3,0,{2,1}
"window" index "window","fenster","ventana"	☛1
"hi" index 2,3,"hi",0	does not work

6. Function Definition

In Earl, Functions are polymorphic; functions can be defined that have the same name, but different numbers of arguments. (This is part of how optional arguments work.) After a function is defined, it becomes an infix operator; the name comes between the arguments if it has two arguments, before it if it has one, and alone if it has none. Because of this, the parser requires that all user defined functions that take arguments be prefixed with ':' when used. (This tells the parser what will be a function before it is declared so that the order of function declarations doesn't matter.) The syntax of a function declaration is:

```
funcDef ::= func nameAndArgs ;
            actionStats
            end.

nameAndArgs ::= nilName
              | unaryName arg1
              | arg2 binaryName arg1
```

External symbols are declared just as they are for cells.

A value can be returned from a function with **return** [*exp*]. If the expression is ommitted, null is returned.

arg1 and *arg2* are just variables local to the function, whose values are initialized to the value obtained by evaluating the expressions given when the function is called. The first *nameAndArgs* defines a function with no arguments. The second defines a function with one argument. When it is used, the name (which is *unaryName*) must be prefixed with ':' and the argument to it is the expression following it. The third defines a function of two arguments. It is used by prefixing *binaryName* with a ':' and the arguments are the expressions on either side of it. For example:

```
func fact x;
return if x<=1 then 1
    else x* :fact x-1
    fi
end
```

```
func x fact y;
return (:fact x)/(:fact y)/:fact x-y
end

func fact;
return "That's a fact!"
end

:fact 5     ☛120
7 :fact 3   ☛35
fact        ☛"That's a fact!"
```

Functions cannot take more than two arguments. If it is necessary you can use a global variable, or pass a list. If you have a function that is of general use, mail a note to earl about it and it can be made standard.

Notice that you must prefix your functions with a ':' if they take arguments, while that does not work for built in functions. This is because the system knows that they are functions, and will always be functions. Since there is no guarantee like that about user functions, the colon serves to tell the parser how it is being used.

7. System Commands

System commands in general deal with the operating system. Saving and restoring workspaces, running a text editor on cells or functions, etc. All system commands are prefixed with ~ and go to the end of the line. Do not use a semicolon to end the line. The actual command may be abbreviated, and case doesn't matter. (This is the only case in Earl where it accepts abbreviations and is case blind.)

~save *fileName*
: Write the workspace to the named file and continue working. The workspace is ordinary text that can be modified with any text editor.

~suspend
: This writes the workspace to the file '.earlspace' and exits. When Earl starts up, .earlspace is read and executed.

~copy *fileName*
: Read the file, executing any statements and defining any cells or functions that are contained in it.

~clear
: Clear the workspace, that is, remove everything that is currently defined.

~load *fileName*
: Clear the workspace and read the file.

~edit {cell and function names separated with spaces}
: The variable "EDITOR" has the name of your favorite editor. It is run and given the text of the cells and functions named. When the editor finishes, those cells are read in again and replace any previous definitions.

~!*shellCommand*
: Make the shell execute the given command. If there is no shell command, you get an interactive shell to use.

~mark
: From now on, when writing the top level cell, put marks on the comment layer (which is NX in nMOS and SX in cMOS-SOS) to show where the ports and edges of the top level cell are.

~nomark
  Stop marking ports and edges.

~debug
  Toggle debug output from the parser.

CHAPTER 2

Implementation Notes

1. But how does it work?

A good question, and one I often ask myself (along with the related, why doesn't it work?). In this section I will describe how the interpreter works. If you only want to use Earl, skip it; there is nothing useful here.

Beyond knowing C, most important to understand Earl's operation, is understanding the data structures and the routines that create, manipulate, and destroy them. After that, the operation of the parser and code evaluation will be described; then other things as they come to mind.

1.1. Data Structures

Most of the structures thrown around by the system are conformable to the structure called a 'thing'. In the header, earl.h, Thing is defined to have all the data common to each thing. This data includes a byte to label the type of the object, a byte for flags, and a short for the reference count. There are symbolic constants (all caps) to define the values for the type field in earl.h. The various structures are made by typedef and usually have the same name (all lower case) as the symbolic constant type number. The meaning of the flags is usually dependent on the object type, they are described in earl.h pretty clearly. The reference count is the biggest headache around. You must be be very careful with it, otherwise grave disorder results. The correct way to use the reference count is to increment it when you get a new copy of the pointer. When a pointer is to be released, the reference count is decremented. At that point, the pointer must be either passed to a subroutine, returned, or given to eStatFree; it is your responsibility to take care of it. For example, a routine that is called by Eval will be handed one or two arguments that have been popped off of the stack. (These objects can have a reference count of zero.) The routine must free them, store them, or return them. Unfortunately, not all routines are careful enough with the counts and some memory gets lost. It loses memory slowly though, and I haven't put much effort into tracking down all the loss. If you modify Earl, be careful.

(One final thing about reference counts, they don't allow circular pointer structures. For the most part, Earl doesn't use them, but there is an exception: ports. When a port is on two sides of a cell, two separate ports are created, each pointing at the other. To avoid the problems of circles in this loop, that pointer is not counted as a permanent pointer; the reference count is not incremented when it is set.)

The types are symbolically defined in earl.h, the names of the structures are stored in the array typeNames in keytabs.c. Also, some structures are shared for different type numbers, notably thing. This should cause no trouble.

To free an object if its reference count is zero, call eStatFree on it. It will recursively free all structures the object points to. If the reference count is not zero, or the object is null, eStatFree just returns.

Another class of objects are called wood (since they formed the structure of the parse tree while the parser still constructed one). Wooden objects have a line number and character number showing where they were in the current source text. The line and character are used for error reporting.

These are the principal data structures used in Earl:

1.1.1. Number

All numbers are complex. They have an xval and yval, for some operations only the xval is considered.

1.1.2. Atom

An atom is wood (since it could be a niladic operator and need to report an error), and has a null-terminated character string called anam. The space for the string is allocated once the text is known.

1.1.3. List

To avoid allocating lots of little bitty sections of memory, lists are implemented with arrays. The array of pointers to things is called vals, the number of locations used in it is lused, and lsiz tells how long it is. Lists are used for almost everything in Earl. A value table for a cell or function is a list, the parser produces rpn strings in lists, the path for a wire is a list of segments, etc.

'append' takes a list and a thing pointer, makes the thing be the last element of the list and returns the list. Since this can cause the list to grow, and thus move, the returned list must be put back where it came from after appending. If a list with multiple pointers to it moves, bad things will happen for sure. Also, you can append to NULL. To keep from having to allocate all the lists one might want in advance, if you append to NULL, a list is created with a reference count of one to which the thing is appended.

1.1.4. Lval

Since value tables can grow and move during execution, an lval must be able to describe how to find the pointer when it is used, not when it is created. This is done by giving the static address of a pointer to a list (the address of the pointer to the stack, sb, or the address of the pointer to the global value table, globVal.), a count of the offsets, and an array of shorts. An lval is turned into the thing it refers to with the function wantNum, and the address of the thing it refers to (for use in changing the value of a variable) with the function wantLoc. wantLoc implements copy on write (assuming that the only reason you want the address of a thing is to change it). When it is traversing the lists looking for the desired address, if it notices a list with multiple references, it makes a copy of the list with a reference count of one and replaces it on its way down. It also extends lists when the lval indexes beyond the end.

1.2. Parsing and Evaluating

1.2.1. Input

Input is in the module eio.c When reading in, it handles typing the prompts, and saving the input text for later. If an error is seen during the parse, the current location and error message is saved until the end of the line is read so a complete line can be given with the error message. Up to five errors can be saved in this way, after that it prints the errors that it has seen, so if there is an exceptionally foul line, it may be printed before it is completed in the buffer.

To re-direct the input, the routine pushToFile will return the old input file, and set the current file to its argument. doWork can then be called to read until the end of that file, to effect a 'push', or work can continue, to just change the input file.

1.2.2. Lexical Analysis

The lexical analyzer, in lexanal.c, recognizes comments and special characters. If it reads an identifier, it calls isKey, in symtab.c, to find out if it is anything special. isKey will return four types of results. If it returns zero, the word is unknown and assumed to be a user variable or function. The token type given to the parser is IDENTIFIER, and the value is an atom with that name. In all other cases, the value returned from isKey is a structure with the function number and a pointer to two functions that will handle it. If isKey returns a positive number, the word is a keyword, and the token type is the value returned. If the value is from -1 to -999, it is a standard function that has a precedence low enough so that it can receive lists as arguments. Otherwise, the return was less than -1000, and it is a standard function that has a precedence so that its arguments can be expressions, but not lists. (for example 'to').

The lexical analyzer also detects 'system commands' beginning with '~'. It reads the following word and calls sysParse to handle them right away; the parser has nothing to do with these.

1.2.3. Parsing

The parser is a yacc parser in parse.y It has several shift/reduce and reduce/reduce errors. Many of these are irrelevant, being impossible to hit anyway (for instance, on the error token when it is impossible to have an error), and the others are between which of two possible error messages to give. It seems to pick a good message, so I don't worry about it too much.

As work proceeds in the parser, sections of code are built up and concatenated together to produce an rpn string. The function that concatenates them together, catOp, is in support.c and takes a variable number of arguments. The first one tells how many other arguments there are, and the others are code fragments to be concatenated together. The concatenation is done by getting a list big enough, taking the first code fragment if it is big enough already, then copying the pointers in. If a fragment was a list, then all the reference counters of the copied things are incremented, and the list is freed. Otherwise, the thing itself is kept and nothing is freed.

1.2.4. Evaluating Code

The routine Eval in earl.c takes the rpn strings produced and evaluates them. It is basically a simple no-address stack machine. When a number is seen, it gets pushed. Atoms get the corresponding lval pushed, unless it denotes a niladic function, causing it to be evaluated, or it is a cell, which pushes an instance of the cell. (you can't assign to a cell – you can't get an lval for it.) There are 'opcodes' made using the structure of a thing with different type numbers: dump causes the top of the stack to be popped and saved in 'Last' (used between statements in a list of statements); givenull pushes a NULL (for statements that otherwise produce nothing. There must be something on the stack after all statements so the following dump doesn't remove something unintended); keeplast saves the top of the stack in 'Last' (that's the value @ returns); quote has a pointer to a thing which gets pushed directly (for horiz, vert, and the name to use for extract). Conditional branches take the top of the stack as the condition and change the pc.

When an operator structure is seen, its arguments are popped and the function is called with the number of arguments, the given function number, and the arguments (any unused are passed as null so there are always two pointers passed). The return value from the function is pushed. If the operator needs to evaluate the code from a cell, it passes the cell and the code to execute to BeginCell, which does a Begin and fills values into the new 'activation record'. For a function, it passes the proc pointer to BeginProc, which also does a Begin and fills in the activation record. Activation records contain pointers to the current source, a pc, a pointer to the current cell or proc, and so on. After the Begin, Eval is called to execute the code that was set up. Eval returns the top of the stack after the code is done; this value is only used at the top level. After Eval is finished, the activation record is removed with End. End checks to see if it is finishing a procedure, when it must dig around the value table and get out the return value from the procedure. It frees up any extra stuff left on the stack and returns.

Eval also knows about for loop constructs. The range of the loop is evaluated, followed by the forbegin opcode. The range is saved on a for-stack, and the code to the following forend opcode is executed once for each value in the range. The forbegin opcode has a pointer to the controlled variable to set it; the for-stack keeps track of the pc at the top of the stack.

2. Algorithms

2.1. Building a Constraint Graph

A constraint graph is made from a list of nodes, and a list of constraints referencing those nodes.

A node in the graph represents a set of ports that move as a unit in one dimension. The coordinates of a port are figured by adding an offset to the position of the nodes it references.

A constraint in the graph indicates by how much the position of the end node must be larger than the position of the start node. There can be only one constraint from a node to any other node, and no constraints from a node to itself. Positions are assigned to nodes to minimally satisfy the constraints.

The constraint graph of a leaf cell is built out of the constraints given by the designer. A constraint specified to be an exact distance, like xcon B |=5| C, replaces the x-node of C throughout the graph with the x-node of B, adding 5 to the offset. B and C then have the same x-node, but C is be offset by 5. A constraint specified as greater than some distance, like xcon C |>6| D, creates a constraint from the x-node of C to the x-node of D with a minimum separation being the x offset of C plus 6 minus the x offset of D. If there already is a constraint on these nodes, the new distance is compared to the old and only the larger distance is kept. So xcon C |>6| D; xcon C |>7| D; will leave only the second constraint in the graph. A constraint specified as less than some distance, like xcon D |<5| E, is treated exactly as the constraint xcon E |>-5| D.

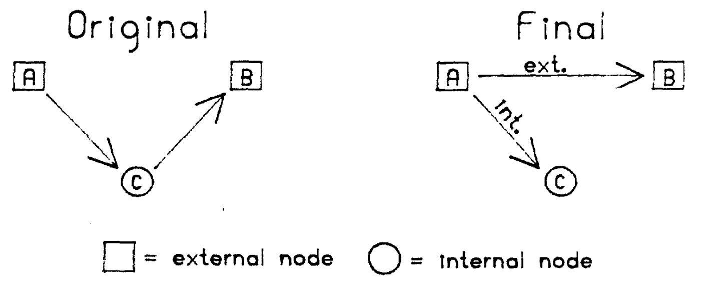
The constraint graph of a composition cell is built out of the constraints on the external ports of the subcells. When a subcell is drawn into the composition cell, the subcell must first calculate its external constraints (which is described in the next section). Then a copy of its external graph is built into the composition cell by creating nodes, ports, and constraints and linking them together to reflect the rotation and mirroring of the subcell. (see the routine doDraw in appendix C.) When that is finished, the graph from the subcell is disjoint from the graph originally in the constraint cell. They are joined by constraining the connecting ports of subcells to be at the same location.

2.2. Extracting External Constraints

To get the external constraints for a cell, first the complete graph must be built, as described in the previous section. The completed graph is then processed to divide it into internal and external parts. The external graph only has constraints on the external nodes; the internal one has constraints with internal and external nodes. This division reduces the number of nodes in a graph by hiding nodes within cells. Since there may be a path in the full graph from one external node to another through an internal node, Earl must find all such paths and make a constraint between the two external nodes that is the most restrictive of all the combined paths between them. This is done with a process that is similar to transitive closure.

```
for each internal node
   for each constraint that starts at this node
      for each constraint that ends at this node
         make a constraint between the other nodes in the
         constraints with where the minimum separation is
         the sum of the separations
      if any were found ending at this node, remove the
      constraint from this node
```

(see the routine tclose in appendix C.) This differs from transitive closure in that it is not applied to all the nodes, and that it removes constraints from the graph. The constraints that are removed are no longer necessary, since all the information that they had is contained in the constraints that were created. For example, suppose there are two external nodes, A and B, and an internal node c; and there is a constraint from A to c and one from c to B. The outer loop will be applied only to c. The next loop will find the constraint from c to B. The inner loop will find the constraint from A to c, and produce a constraint from A to B. The constraint from c to B will be removed. The constraint from A to B will be passed up the hierarchy, guaranteeing that there is enough space between them to satisfy both of the original constraints, from A to c and from c to B. Since positions are assigned to minimally satisfy the constraints, c will be as close to A as the constraint allows, guaranteeing that it satisfies the, now removed, constraint from c to B also.



Extract External Constraints

Cycles in the constraints are resolved with this process. If there are external nodes in the cycle, all the internal nodes will be removed from it and a smaller cycle will be passed up the hierarchy. Otherwise, the loop will be reduced until a constraint appears from a node to itself. If the spacing in that constraint is less than or equal to zero, the constraint is always true, and is thrown out. If the spacing is greater than zero, the constraint is never true and indicates an error in the constraint graph. Two of the nodes of the loop are known, the one in the constraint and the internal node, and can be given to the designer to help in locating the problem.

A property of this algorithm is that when it terminates each internal node may start constraints, or end constraints, but not both. The ones that end up starting constraints are those that had no constraints ending on them when they were processed. If any constraints did end on a node, all of the constraints starting with that node would be remooved. Once removed, no other constraints can be created that start with that node, since the new constraints always start with a node that started a constraint, and end with a node that ended a constraint. This division means that there are no cycles left in the internal graph. Also, if an internal node only begins constraints then it originally had no constraints ending on it, or it was part of a cycle of internal nodes that had no constraints ending on it. (Only one of the nodes in the cycle will start constraints. The other members of the cycle will be constrained by the one that starts constraints.) In either case, it is correct, though poor, for that node to get a position that is infinitely less than the position assigned to any external node.

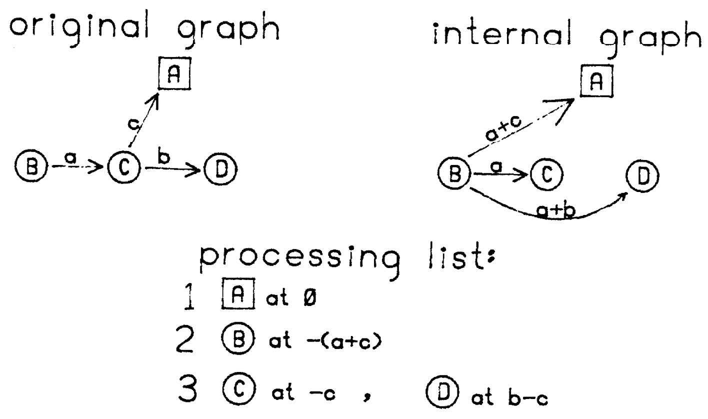
2.3. Assigning Coordinates

After building the constraint graph for the top level cell, Earl goes on to give nodes positions. All the external nodes in a cell will receive coordinates from 'on high' that cannot be changed, and are put into a list of nodes (the initial order doesn't matter). Then

```
while the list isn't empty
   for each internal constraint connected to the first node in the list
      make sure it is satisfied by either assigning a coordinate
      to the other node, moving the coordinate that it already had,
      or just leaving it alone.
      if the other node has not been in the list yet
         put it at the end of the list
   remove the first node from the list, making the second the new first.
```

If the list ever becomes empty and there are still nodes left to process, pick a node that hasn't been in the list, give it some random coordinate (zero is pretty random), and process it.

This algorithm is guaranteed to satisfy all the constraints in the original graph when given an internal graph of the form produced by the external constraint extraction and the external nodes are assigned positions that satisfy the transitive closure of the initial constraints. After all the external nodes have been removed from the list, all the constraints between internal and external nodes have been satisfied. Internal nodes that started constraints have a position that is the minimum of the positions calculated from the position of an external node minus the minimum spacing (though lower would still be correct). Internal nodes that end constraints have the position determined by the maximum of the positions calculated from the position of an external node plus the minimum spacing. Its position satisfies all original constraints starting on it, since those constraints were passed back to constrain the positions of the external nodes, leaving enough room between the external nodes for both constraints. Next the internal nodes are processed. The only constraints that still haven't been processed are those from internal nodes to internal nodes. If they aren't already satisfied, they will be satisfied either by moving a node that starts constraints to a lower position, or moving a node that ends constraints to a higher position. The first is always correct. The second case will be correct, since any constraint which might be violated by such a move, will have been transformed into a constraint starting on the first node so there will be enough room to make the move without violating the constraint. That is, if node A moves node c closer to node d, and there was a constraint from c to d, there is also a constraint from A to d that will make sure the constraint from c to d is satisfied. Although the result may be different if nodes are processed in a different order, each different solution is correct.



Coordinate Assignment

The algorithm must be modified slightly for top level cells, since they do not receive positions for external nodes from higher up in the hierarchy, and since this algorithm does not handle cyclic graphs. What is done, is to redefine external nodes to be the nodes that have no constraints ending on them, and apply the external constraint extraction algorithm. This definition of external nodes guarantees that no external constraints will be generated and all the nodes can be given arbitrary positions. At that point, the ordinary coordinate assignment algorithm can be applied.

Once all the nodes have a position, the coordinates of the ports can be found by adding the offset to the position of the referenced nodes. Earl then computes the positions of the external nodes of a subcell. It finds the coordinates of the port called 'sw' (which is put in by Earl to indicate the southwest corner of the subcell) and computes the offset of each node relative to it. Using the transformation that drew the cell, the relative spacings are transformed back to find where the corresponding node is in its original cell. (Earl makes a list of the positions of the external nodes in a 'pvector'. The pvector gets used later to determine if the cell was stretched in a different way, or if a previous CIF cell represents it.) These positions determine the positions of the external nodes of the subcell, and Earl recurses until all the cells have been drawn.

2.4. The Path of a Wire Using Arcs

The path of a wire is specified with a list of points with associated radii. A radius of zero indicates that the center line of the wire is to include the point. A non-zero radius indicates that the path must be at least that distance away from the point. A path is scanned before it is written to remove any points that the path will not need to curve around. The algorithm keeps a sorted list of line segments, in order of direction, that limit the region containing the possible path. The input is the path of a wire, and it outputs another, possibly shorter, list of points and radii. The algorithm is:

```
Output the first point. (it must have a zero radius)
for each following point
   if the radius is >0 (counterclockwise)
      while the list is not empty
         compute the path from the first point on the top segment
         tangent to the current point.
         if the direction from this new ray to the top ray is
         negative (clockwise), exit the loop.
         if the final point on the top ray has a negative radius
            output that point
         remove the top segment from the list
      if the list is empty
         put the ray from the last point output to the current
         point on top of the list
      else if the radius of the last point of the top ray is >0
         put the ray from the last point on the top ray to the
         current point on top of the list
      else (the radius of the last point on the top ray must be <0)
         put the ray from the first point of the top ray to the
         current point on top of the list.
   else if the radius is <0 (clockwise)
      much the same as above, but work from the bottom of the list
      up, the signs of each comparison is reversed, and the ray
      gets added to the bottom of the list.
   else (the radius of the current point must be zero)
      do the loop for the case when the radius is positive
      do the loop for the case when the radius is negative
      output the current point (the list must now be empty.)
```

(this is based on the circle obstacles algorithm in [Tompa 80].) Earl treats circles as dodecagons, since CIF does not have arcs.

APPENDIX A

Quick Reference

1. Standard Identifiers

The standard identifiers are in the symbol table and may be redefined, but most of them should not be redefined. When they are used in functions or cells they must be declared external.

nmos and cmos
: Functions to declare the technology.

lambda
: The scale of the project measured in microns. (lambda and the layers for the current technology are automatically declared external to cells.)

EDITOR
: The name of the editor to use with the ~edit command

PLOTARGS
: Command line arguments to the cifplot program.

merge, emerge, nmerge, smerge and wmerge
: Flags to the composition functions.

printgraph
: Prints the constraint graph in an almost readable form. This is for debugging only.

getchar, getstring and geteval
: Input a character, a string, or an expression from the input file.

rebuild
: Cause the constraint graphs to be rebuilt when they are needed again. This happens automatically when a cell is parsed (it assumes that you are redefining a cell and could have changed the constraints in it). This would be necessary if you changed a variable that determines the number of cells or something like that.

2. Reserved Words

The following conventions are used to describe the functions. **literals,** [*zero-or-one-of-these*], {*zero-or-more-of-these*}, *one-of-something-like-this*, ☛*the-result*. When describing many similar functions, **fn** will stand for any one of them.

The reserved words are:

abort, and, andif, appfile, ascii, box, ccwmiss, ceil, cell, check, cifp, colorplot, colourplot, connect, constr, cos, cvs, cwmiss, dm, do, dp, dpe, dpn, dps, dpw, draw, drawcif, each, ef, elecnode, end, else, external, fcifp, fi, floor, for, func, geom, group, hcompose, horiz, if, iie, iin, im, indix, infile, ipe, ipie, ipin, ipn, ips, ipw, len, make, max, mcifp, min, miss, mx, my, od, or, orif, outfile, named, not, plot, pm, point, points, polygon, ports, print, rep, repeat, return, rot, seg, sin, sleep, strlen, substr, then, to, until, vcompose, vert, while, wire, without, xcon, ycon

Layout functions

group-name **group** *names-of-ports.*
Create a group with a list of ports.

points *point-names.*
Creat internal points. (also point.)

dm, dp, dpe, dpn, dps, dpw, iie, iin, im, ipe, ipie, ipin, ipn, ips, ipw, pm:
fn *points* ☛*same-points.* Connect layers at the points in the list and return the list unchanged.

mx, my: **fn** *inst1* ☛*inst2.*
Mirror x (x coordinates negated), or y (y coordinates negated).

inst1 **rot** *num* ☛*inst2.*
Rotate inst1 num times 90° counterclockwise.

[*layer*] **box** *pt1,pt2.*
Make a box on the given layer (default is the previous layer) that has pt1 and pt2 as opposite corners.

[*layer* | *layer,width* | *width* | *old-path*] **wire** (*points* | *old-path*)
☛*current-path.* Make a wire on the layer (defaults to the last layer used) of the given width (minimum width for the layer), or set the current wire to the old wire. Append the points to the center line. To just set the current path without adding points, use no left argument and the old path is the right argument.

[*layer* | *old-path*] **polygon** (*points* | *old-path*)
☛*current-path.* Make a polygon on the layer (defaults to last layer used), or set the current path to the given path. Append the points to the path, the edge of the polygon. To just set the current path without adding points, use no left argument and the old path is the right argument.

miss *point, dist.*
Ensure that the current path misses the point by at least dist at this part of the path. (if the wire loops back it still can get too close.) Positive distances are counterclockwise, negative are clockwise.

cwmiss, ccwmiss: **fn** *point, dist*
Just like miss, except that cwmiss makes a clockwise turn, and ccwmiss makes a counterclockwise turn.

seg *points.*
Append more points onto the path of the current wire. Only necessary after a miss.

draw *inst1* ☛*inst2.*
Build the constraint graph of inst1 into this cell. When this cell is output, inst1 will also. The ports and groups of inst2 are accessible.

port-list1 **connect** *port-list2.*
Connect the ports in list1 to those in list2. The ports must come from instances which have been drawn.

hcompose, vcompose: [*merging-parameter*] **fn** *instance-list.*
Cause the current cell to be a composition of the listed instances. *merging-parameter* is one of **merge** , **emerge** , **nmerge** , **smerge** , or **wmerge** .

insts **named** *names* ☛*insts*
set the name of the instances so the circuit extractor can distinguish the nodes.

[*source-file*] **drawcif** *cellnam,mx,rot,location.*
   When this cell is output, include a CIF call to the cell named *cellnam* found in the source file (which defaults to ~earl/Lib/nmos or cmos). Avoid using this.

[*layer*] **elecnode** *point,nodeName*[*,nodeType*]
   Tell Tom Hedges's circuit extractor the name of an electrical node. The layer defaults to the most recently used layer.

Support Functions

check *cell.*
   Check a cell for out of order ports. Does not always detect them! (it's better than nothing.)

[*file-name*] **make** *cell.*
   Output CIF for the cell to the named file. The file defaults to *cellName*.**cif**

[*cifp-options*] **cifp** *cell.*
   Compute CIF for the cell and pipe it to cifp. fcifp does the same thing, but lets cifp subprocesses pipeline. mcifp is like fcifp, but it beeps after finishing plotting.

plot *cell.*
   Compute CIF for the cell and pipe it to cifplot(1) to make a black and white print. Put arguments to cifplot into the variable PLOTARGS.

colourplot *cell.*
   Compute CIF for the cell and pipe it to cifplot(1) to make a color print. Put arguments to cifplot into the variable PLOTARGS. (also colorplot.)

or, and, orif, ||, andif, &&: *exp1* **fn** *exp2*☛*exp3.*
   or and **and** evaluate both arguments; the others evaluate the left one, and evaluate the right one only if necessary. True is anything non-zero, False is zero. There are no bitwise operations; there are no integers.

not *exp1*☛*exp2.*

cos, sin: **fn** *exp1*☛*exp2.*

start **to** *stop*☛*list.*
   generate a list from start to stop.

list1 **without** *list2*☛*list3.*
   remove each element of list2 once from list1 if it is there.

len *exp1*☛*exp2.*
   give the length of a list, 1 if it isn't a list, or 0 if the argument is null.

strlen *exp1*☛*exp2.*
   give the length of a string, 0 if it is null, and -1 if it isn't a string.

max, min: *exp1* **fn** *exp2*☛*exp3.*
   give the number with the larger (smaller) x component, or the string with the larger (smaller) value. Uses corresponding elements of lists.

ceil, floor: **fn** *exp1*☛*exp2.*
   Convert the x and y components to integers: largest integer less than the component or smallest integer larger than the argument.

exp1 **rep** *num*☛*exp2.*
   Repeat exp1 num times.

sleep *num.*
   suspend execution for num seconds.

infile, outfile, appfile: **fn** *fileName* ☛*flag*.
: open the file named for input, output, or append to the file. Returns 1 (true) on success and 0 (false) on failure.

print *exp*.
: print the value of the expression on the output file.

abort *exp*.
: print the value of the expression on the terminal and abort execution.

cvs *exp1* ☛*exp2*.
: convert lists of numbers to lists of strings.

exp1 **substr** *exp2* ☛*string*.
: *make a string by selecting characters from the string corresponding to the numbers in the second expression.*

ascii *exp* ☛*exp*.
: turn a number into a string with one character, and a string into a list of numbers whose values correspond to the ascii value of the characters. Turns consecutive numbers in a list into a string, strings in a list into a bunch of numbers in the result list, and a list gets processed recursively by ascii and becomes an element of the result list.

items **index** *list* ☛*exp*.
: result is the same shape as *items* where each element is replaced by the index of the first place it is found in the list, or zero if it is not found.

Obsolete Words

These are still in, but will be removed. Use none of them: **each, ports**

Syntax

This is a simplified version of the syntax of earl. (The actual syntax is in the file parse.y)

statement ::= *actionStat* ;
| *defineStat*
actionStat ::= *exp*
| *whileStat*
| *repeatStat*
| *forStat*
| *externalDecls*
| *confunc constraints*
| **return**
| **return** *exp*
actionStats ::= *actionStat* {; *actionStat*}
exp ::= *Identifier*
| *constant*
| *cellParameterization*
| "(" [*exp*] ")"
| "{" *exp* "}"
| *exp* "[" *exp* "]"
| *exp* . *Identifier*
| *unaryOp exp*
| *exp binaryOp exp*
| **horiz** *exp*
| *exp* **horiz** *exp*
| **vert** *exp*
| *exp* **vert** *exp*
| **if** *exp* **then** *actionStats* {**ef** *exp* **then** *actionStats*}
[**else** *actionStats*] **fi**
cellParameterization ::= *Identifier* "(" *exp* ")"
whileStat ::= **while** *exp* **do** *actionStats* **od**
repeatStat ::= **repeat** *actionStats* **until** *exp*
forStat ::= **for** *Identifier* := *exp* **do** *actionStats* **od**
externalDecls ::= **external** *Identifier* {, *Identifier*}
confunc ::= **xcon** | **ycon**
constraints ::= *exp* {"|" *relop* [*exp*] "|" *exp*}
defineStat ::= *funcDef* | *cellDef*
funcDef ::= **func** *nameAndArgs*
actionStats **end**
nameAndArgs ::= *funcName* ;
| *funcName arg1* ;
| *arg2 funcName arg1* ;
cellDef ::= **cell** *cellName* ["(" *paramName* {, *paramName*} ")"] ;
actionStats
[**constr** *actionStats*]
[**geom** *actionStats*]
end

APPENDIX B

Captain Earl Secret Decoder Ring

Constraint Messages

Captain Earl uses a secret format to identify nodes in the constraint graph when giving error messages. First the node number is given. Node numbers are basically meaningless; they are assigned sequentially from zero when Earl starts up. Next, each port connected to it is listed. If the port is offset from the node, the offset is listed first. Then, the name of the port in the leaf cell that declared it. If the error occurs in a composition cell, the name will be followed with the number of the subcell within the composition that the port came from. Finally, which constraint graph, x or y, the node is in.

```
node number 15:    =a.x,   -3=b.x,
```

This shows that the x coordinate of *a* is the position of node 15, and the x coordinate of *b* is the position of node 15 minus 3.

```
node number 234:    +13=in.1.y,    +13=out.2.y,
```

This shows that the port named *in* in the first instance, has the same y coordinate as *out* in the second instance. The ports *sw*, and *ne* indicate the southwest and northeast corners of every cell.

A self constraining node occurs when a constraint is generated from a node to itself, with a positive minimum separation. Such a constraint requires that the position of the node be greater than its position. They can be created directly with a statement like:

```
xcon a |=3| b |>-2| a;
```

This is detected when the constraint is made. The error message tells which port it was trying to constrain when the error was detected (it will be *a* in this case), and what the result would be if the constraint were made. This error can be fixed by just looking at the constraints and making them consistent.

Self constraining nodes can also be made by connecting two cells together. For instance,

```
cell foo;
west group a,c
constr
ycon a |=1| c
end

cell bar;
east group b,d
constr
ycon b |>2| d
end

boom horiz bar,foo;
```

When the composition *boom* builds its constraint graph, it will try to connect the ports together. At first a and c have the same y node because of the constraint in cell foo. After connecting a and b, the ports a, b, and c will all have the same y node. When c gets connected to d, the constraint from b to d cannot be satisfied, since c is in the same node as b. That constraint will be shown, along with the named nodes. This sort of problem often occurs when ports are not connecting correctly. Check the node lists for ports that should not be in the same node.

An illegal constraint loop is a collection of nodes that form a loop, A constrains B constrains C constrains A, such that the sum of the minimum separations around the loop is positive. This is detected when extracting the external constraints for a cell, but only after the loop has been reduced to two nodes. The error message will identify the two nodes it has left and offer to print the graph. Figuring out the cause of the loop is not difficult when it happens in a leaf cell. When it happens in a composition cell, it is likely to come from connecting ports improperly. For example:

```
cell foo;
west group a,c
constr
ycon a |>3| c
end

cell bar;
east group b,d
constr
ycon d |>2| b
end

boom horiz bar,foo;
```

In cell bar, b comes before d in east, but d is constrained to come before b. The best help for this situation is the function **check** *inst*. Check tries to see if ports are constrained to have a position that is not in the same order as they are declared in the groups. It only detects if there is a constraint that directly puts them out of order, but that is often the case.

If that doesn't point out the error, you can look at the graph.

The Constraint Graph

You can get the constraint graph by answering yes when it asks if you want one, or by putting the **printgraph** at the point where you want to see what you have. The "graph" begins with a listing of the ports in the cell. The name of each port is listed, followed with an "*" if the port is accessible to the outside. (Unnamed ports are created to have something to constrain for groups that would otherwise have no ports. These ports can be ignored.) An edge is listed if the port was on an edge; either of this cell or of a subcell. If a port was on the edge of a subcell, it will have an edge letter, but no "*." Next comes the numbers of the x and y nodes that this port is in, followed with the port's offset from its nodes. When the graph is printed in the layout section of a cell, the actual coordinates of the port are shown. Sometimes a number will follow this, indicating that the port is linked across the cell to another port of the same name. For the above example, the port list would look like:

```
ports:
sw*          2,3     0#0
ne*          0,1     0#0
sw           2,3     0#0
ne           13,1    0#0
b       E    13,16   0#0
d       E    13,17   0#0
sw           13,3    0#0
```

and so on.

The next section just lists the nodes in the cell, along with a bunch of debugging information for me.

The last sections show constraints. There may be up to six lists showing the Original, Internal and External graphs in X or Y. Each constraint is listed as

```
from -- minsep --> to
```

showing the numbers of the from and to nodes, and the minimum separation between them.

In my experience I have found that the fastest way to fix this kind of problem is by using check, looking carefully at the ports that have been joined into the nodes, and thinking about it. It is usually very difficult to find the error in the graph.

Circle in a Circle

Circle in a circle means that somewhere on the path of a wire, you wanted to find a tangent that didn't exist; either from a point inside a circle to the circle or from one circle to one that it intersects.

Often this error is detected when you are building the path, in which case Earl will show the source line that is in error. To fix this, just move the points farther apart. I often don't know how far to move them, so I use **print** *exp* to show where the points are.

This may also be detected between two non-adjacent points on the path when writing CIF because of the properties of the algorithm that puts in the arcs. When this happens you are given the coordinates of the centers, the radii, and which cell it happened in. Fixing the problem involves either moving the points farther apart, or adding a point to the path forcing it to choose a route that works.

Whenever it happens, marks are put on the comment layer to show where the centers of the circles are, and a circle is drawn showing how far apart they must be.

APPENDIX C

Selected Routines

doDraw

```
/* draw an instance.  i.e., when this cell gets drawn, so will the instance */
thing *doDraw(n,fun,arg,arg2)
int n,fun; thing *arg,*arg2;
{   instance *insi;
    BOOLEAN swapx,swapy,flip;
    list *vix,*viy,*nl,*icl,**ocl;
    port *pi,*pj;
    int i;
    constr *cni,*cnj;

    /* get the arguments and check for errors */
    ASSERT(n==1 AND fun==1);
    if ((arg=pusharg(arg,"no instance"))->typ!=INSTANCE)
        runError("this is a %s, not an instance",typeNames[arg->typ]);
    insi=(instance*)copyThing(arg);
    if (top->curCell==NULL){
        runWarn("warning--draw is not needed outside of a cell definition");
        return((thing*)insi);
    }
    if (insi->mydef->extra)
        runError("recursive draw of cell %s while in cell %s",
                 insi->mydef->cname,top->curCell->cname);
    /* keep a list of drawn subcells in the cell */
    top->curCell->insts=append(top->curCell->insts,(thing*)insi);
    /* if the x dimension gets negated */
    swapx=(insi->rot>=2) ^ (insi->mx!=0);
    /* if the y dimension gets negated */
    swapy=(insi->rot==1) OR (insi->rot==2);
    /* if the dimensions will be swapped */
    flip=insi->rot&1;

    /* compute the external constraints for the subcell if they aren't known */
    getExt(insi->mydef);

    /* get vectors with as many nodes as mydef had externals */
    vix=newList(insi->mydef->exnnum);
    for (i=insi->mydef->exnnum; i>0; --i)
        vix=append(vix,(thing*)newNode(!flip));
    viy=newList(insi->mydef->eynnum);
    for (i=insi->mydef->eynnum; i>0; --i)
        viy=append(viy,(thing*)newNode(flip));


    /* copy mydef's external ports into the current cell's graph */
    /* as this is done, give it the proper nodes, adjust the offset
```

```
     * and the edge number to account for the transformation, and copy
     * the links across the cell */
    nl=insi->mydef->alias;
    for (i=0; i<nl->lused AND (pj=(port*)nl->vals[i])->ext; i++){
        if (pj->edge>=WEST AND pj->other!=NULL AND pj->other->ext)
            continue;
        /* enter the port into this cell's list of ports */
        safeAlias(pi=newPort(pj->pname));
        /* make pi in this cell look as pj did in the subcell */
        transformPort(pi,pj,insi,vix,viy,swapx,swapy,flip);
        if ((pj->edge==EAST OR pj->edge==NORTH) AND pj->other!=NULL
                    AND pj->other->ext){
            /* if the port is linked to one on the other side, make the other */
            safeAlias(pi->other=newPort(pj->pname));
            pi->other->other=pi;
            transformPort(pi->other,pj->other,insi,vix,viy,swapx,swapy,flip);
        }
    }

    /* copy the external constraints from mydef into the current cell */
    for (curdimx=TRUE; curdimx>=FALSE; --curdimx){
        nl=(curdimx^flip)? vix : viy;
        icl=(curdimx^flip)? insi->mydef->excon : insi->mydef->eycon;
        ocl=(curdimx)? &top->curCell->oxcon : &top->curCell->oycon;
        if (icl!=NULL){
            for (i=icl->lused-1; i>=0; --i){
                cnj= (constr*)icl->vals[i];
                cni= ((curdimx^flip)?swapx:swapy) ?
                    newConstr(
                        (node*)nl->vals[cnj->tnd->num],
                        cnj->minsep,
                        (node*)nl->vals[cnj->fnd->num]) :
                    newConstr(
                        (node*)nl->vals[cnj->fnd->num],
                        cnj->minsep,
                        (node*)nl->vals[cnj->tnd->num]);
                *ocl=append(*ocl,(thing*)cni);
            }
        }
    }
    eStatFree((thing*)vix);
    eStatFree((thing*)viy);
    return((thing*)insi);
}
```

tclose

```
/*  nlist is the list of nodes, sorted so that the external ones are first;
 *  clist is the list of constraints, random order;
 *  icons gets a list of the internal constraints
 *  econs gets a list of external constraints.
 */
tclose(nlist,clist,extnodnum,icons,econs)
list *nlist,*clist,**icons,**econs;
int extnodnum;
{   register int j,k,nk;
```

```
register NUM *consts;
int i,n,nsq;
constr *ci;
NUM *dp:
double sep;
int found,ni,nsq;

if (clist==NULL OR nlist==NULL) return;
n=nlist->lused;
if ((consts=(NUM*)malloc((unsigned)((nsq=n*n)*sizeof(NUM))))==NULL)
    runError("out of room in tclose");

/* fill the array with illegal floating point values, to differentiate
 * between arcs and no arcs. */
nanfill(consts,nsq);

/* put the constraints into the array */
for (i=clist->lused-1; i>=0; i--){
    ci=(constr*)(clist->vals[i]);
    consts[n*ci->fnd->num + ci->tnd->num] = ci->minsep;}

/* here's the transitive closure work */
/* for each internal node */
ni=nsq;
for (i=n-1; i>=extnodnum; i--){
    ni-=n;

    /* for each constraint from this node */
    for (j=n-1; j>=0; j--){
        if (isnum(consts[ni + j])){
            found=FALSE;

            /* for each constraint to this node */
            nk=nsq;
            for (k=n-1; k>=0; k--){
                nk-=n;
                if (isnum(consts[nk + i])){

                    /* make a constraint between the other nodes */
                    found=TRUE;
                    sep=consts[ni+j] + consts[nk+i];
                    if (k==j){
                        /* a loop was closed.  If it illegal, give an error
                         * message */
                        if (sep>0.05)
                            findLoop((node*)nlist->vals[i],
                                     (node*)nlist->vals[k]);
                    } else {
                        dp= &consts[nk + j];
                        if (NOT isnum(*dp) OR (*dp < sep)) *dp=sep;
                    }
                }       /* end of constraint to this node */
            }
            if (found) nanfill(&consts[ni + j],1);
        }       /* end of constraint from this node */
        }
    }       /* end of transitive closure */

    Ethrow((thing**)icons);
    Ethrow((thing**)econs);

    /* now put the constraints into the lists */
    for (i=n-1; i>=0; i--)
        for (j=n-1; j>=0; j--)
            if (isnum(consts[i*n + j])){
                ci=newConstr((node*)nlist->vals[i],
                             consts[i*n + j],(node*)nlist->vals[j]);
                if (i<extnodnum AND j<extnodnum)
                    *econs=append(*econs,(thing*)ci);
                else
                    *icons=append(*icons,(thing*)ci);
            }
    free((char*)consts);
}
```

coordiNode

```
/* Assign positions to the nodes */
coordiNode(nl,cl,snum)
list *nl,*cl;   /* list of nodes and list of constraints */
int snum;       /* number of external nodes */
{   list *wlist;    /* processing list */
    int i;
    constr *ci;
    node *nni;

    /* put the external nodes into the processing list */
    wlist=newList(nl->lused);
    for (; snum>0; )
        wlist=append(wlist,nl->vals[--snum]);

    /* snum is the index of the first element, to avoid copying */
    while (snum<nl->lused){
        if (snum>=wlist->lused){
        /* the list is 'empty' but there are still nodes without positions */
            for (i=0; i<nl->lused; i++){
                if (NOT nl->vals[i]->set){
                    wlist=append(wlist,nl->vals[i]);
                    ((node*)nl->vals[i])->pos=0;
                    nl->vals[i]->set=TRUE;
                    break;
                }
            }
        }
        /* get the first node */
        nni=(node*)wlist->vals[snum];
        if (cl!=NULL)
            for (i=cl->lused-1; i>=0; i--){
                if ( (ci=((constr*)cl->vals[i]))->fnd == nni){
                    /* nni starts the constraint */
                    if (ci->tnd->set){
                        /* the other node has a position, make sure the
                         * constraint is satisfied */
                        if (ci->tnd->pos < ci->fnd->pos + ci->minsep)
                            ci->tnd->pos = ci->fnd->pos + ci->minsep;
                    } else {
                        /* the other node doesn't have a position, give it a
                         * legal one */
                        ci->tnd->pos=ci->fnd->pos + ci->minsep;
                        ci->tnd->set=TRUE;
                        /* and put it on the end of the processing list */
                        wlist=append(wlist,(thing*)ci->tnd);
                    }
                } ef (ci->tnd == nni){
                    /* nni ends the constraint */
                    if (ci->fnd->set){
                        /* the other node has a position, make sure the
                         * constraint is satisfied */
                        if (ci->fnd->pos > ci->tnd->pos - ci->minsep)
                            ci->fnd->pos = ci->tnd->pos - ci->minsep;
                    } else {
                        /* the other node doesn't have a position, give it a
                         * legal one */
                        ci->fnd->pos=ci->tnd->pos - ci->minsep;
                        ci->fnd->set=TRUE;
                        /* and put it on the end of the processing list */
                        wlist=append(wlist,(thing*)ci->fnd);
                    }
                }
            }
        /* move the index to the front of the list */
        snum++;
    }
    eStatFree((thing*)wlist);
}
```

References

[Mead&Conway 1980]
Mead, C.A., Conway, L.A. "Introduction to VLSI Systems" Addison-Wesley, 1980

[Mead]
Mead, C.A. Private communications

[Rowson 1980]
Rowson, J.A. "Understanding Hierarchical Design" Ph.D. Thesis, April 1980 California Institute of Technology

[Tompa 1980]
Tompa, M. "An Optimal Solution to a Wire Routing Problem" Preliminary version, Technical Report 80-03-01 University of Washington