THE UNIVERSITY *of* EDINBURGH

Generating Programming Environments with Integrated Text and Graphics for VLSI Design Systems

George Alexander McCaskill

Ph D

University of Edinburgh

1986

Abstract

The constant improvements in device integration, the development of new technologies and the emergence of new design techniques call for flexible, maintainable and robust software tools. The generic nature of compiler-compiler systems, with their semi-formal specifications, can help in the construction of those tools. This thesis describes the Wright editor generator which is used in the synthesis of language-based graphical editors (LBGEs). An LBGE is a programming environment where the programs being manipulated denote pictures. Editing actions can be specified through both textual and graphical interfaces. Editors generated by the Wright system are specified using the formalism of attribute grammars.

The major example editor in this thesis, Stick-Wright, is a design entry system for the construction of VLSI circuits. Stick-Wright is a hierarchical symbolic layout editor which exploits a combination of text and graphics in an interactive environment to provide the circuit designer with a tool for experimenting with circuit topologies. A simpler system, Pict-Wright: a picture drawing system, is also used to illustrate the attribute grammar specification process.

This thesis aims to demonstrate the efficacy of formal specification in the generation of software-tools. The generated system Stick-Wright shows that a text/graphic programming environment can form the basis of a powerful VLSI design tool, especially with regard to providing the designer with immediate graphical feedback. Further applications of the LBGE generator approach to system design are given for a range of VLSI design activities.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Programs and Pictures

The task of designing artifacts, be they electronic circuits, computer programs or suspension bridges, has led to the development of many design notations. Notations allow a designer to formulate a design idea using pen and paper or a computer display without the need to build the object that is being represented. The representation of the design can take many forms and can model the design at many levels of abstraction. Notations are usually characterised by two properties; syntax and semantics. The syntax of a notation is a set of rules describing the primitive elements of the notation and their legal compositions. The semantics of a notation are a set of rules which assign meaning to the structures formed using the syntactic rules. An example of a design notation is a schematic drawing representing an integrated circuit. Here both pictures and text are used to describe the interconnection of logic gates. Notations using pictures are generally at one level of the design hierarchy; a schematic drawing does not include details of logic gate implementation. A schematic drawing is a mixed notation since it often contains textual annotation of its symbols. The introduction of text allows a drawing to refer to other drawings and to label iterated structures with numerals (e.g. a number of wires can be abstracted into a bus).

Textual notations are more abstract than pictures in that they are not directly representative of an object and can be used to express notions such as hierarchy, data abstraction, parameterisation, iteration and expression evaluation. In computer science, textual representations of execution sequences, i.e. programming

languages, are abundant and have been invented at every opportunity. Although attempts have been made at expressing *conventional* programming concepts using pictures [14] [42] this is still very much an undeveloped medium.

The major property of a pictorial representation, concreteness, has both advantages and disadvantages. A human can comprehend a schematic drawing more easily than its corresponding net-list, however, concepts like parameterisation and conditional evaluation are hard to express using pictures.

A further issue in the design and selection of notations is their suitability for processing by computer. The criteria for adopting a CAD approach include:

Implementation Costs Issues here include design time, development time (coding) and verification (debugging). Formal descriptions of notations can lead to greater confidence in correctness and also the use of automatic implementation techniques. Although formal techniques are often used in programming language development, the implementation of graphics systems is generally more ad-hoc.

Machine Resources Notations which use colour pictures require more sophisticated hardware than those using only black and white. Purely textual notations can run on the humblest of hardware. With the current advent of *affordable* graphics workstations, the use of pictorial notations is becoming more prevalent.

Performance At an extreme level of concrete representation, for example a 3-D solid modelling package complete with colour and shading, the amount of computation involved in generating an image generally precludes fast user manipulation of the design. Some notations are distinctly *batch orientated*, i.e. each modification of the design requires a lengthy compilation phase. The use of abstraction mechanisms, e.g. stick models for solids, can significantly improve performance, at the expense of detail.

Adaptability Specifications of systems can evolve with time. In a rapidly developing area, such as integrated circuit technology, software tools have to cope with changing system parameters.

When considering design notations it is important to separate the notions of syntax and semantics. The syntax of a design notation ultimately defines the physical appearance of the design representation as presented to the user on the video terminal or graphics display. The semantic rules define the translations and evaluations to be performed on the structure obtained from the syntactic stage. The effectiveness of the syntax and semantics of a design notation can be measured using the following properties:

Clarity To what extent is the underlying semantics 'intuitively' obvious from the notation's syntax ? How easy is the notation to learn ?

Conciseness How economically (in terms of amount of actual text or graphics) does the notation represent the structures being designed ?

Elegance This is a less tangible property than the others, elegance is often achieved by using simple and general notational constructs which are applied uniformly throughout the design notation. The elegance of a language directly impacts on the ease with which it can both be learned and used.

Malleability This is a property which relates to the notation's suitability for manipulation using a syntax directed editor. Certain syntactic and semantic constructions can be efficiently exploited by such editing systems (as I will show in later chapters).

Flexibility Can the notation adapt to changing system parameters (e.g. new composition rules, new primitives) ?

Security How easy is it to build incorrect constructs ? Can errors in a design be detected during the *static* analysis phases ?

This thesis is concerned with design notations that combine text with graphics, thus providing the user with the benefits afforded by both representations. I present a method for formally describing these systems and a software tool, the Wright editor generator, which can automatically synthesise programming environments from this specification. Particular emphasis is made on applications in the field of VLSI CAD tools, and a VLSI tool built using the Wright system is described.

The interest and diversity of the problems involved in VLSI have made it a very popular research area. For many researchers it is the combination of different cultures (hardware, software, electronics, language design, graphics, verification, simulation, architecture ...) which makes it so attractive. This mixture has generated the need for a diversity of user-interfaces and translation mechanisms, making use of both linguistic and pictorial notations. This thesis is concerned with providing the means for describing and building those interfaces and translation mechanisms.

1.2 Outline of Thesis

In the next chapter I describe in more detail the need for VLSI design tools and survey both the VLSI and programming environment literature.

Chapter 3 presents the Wright editor generator and gives an introduction to attribute grammars, LALR parsing and structure editing.

In Chapter 4 I run through the development of an example editor using the Wright system, namely Pict-Wright, a picture editor used to build some of the illustrations in this thesis.

Chapter 5 contains the description of a VLSI tool, Stick-Wright, a hierarchical symbolic layout editor which was developed using the Wright system.

In Chapter 6 I present my conclusions and outline areas for future work. A reference manual for the Wright system is included as an appendix.

Chapter 2

VLSI and Programming Environments

2.1 Introduction

In 1981 Carver Mead [55] heralded a new era of technological innovation which he claimed would rival the industrial revolution in its impact. He was referring to the emergence of Very Large Scale Integration technology (VLSI) and the potential for its exploitation. As the number of devices which can be integrated on a single chip continues to grow, the major problem in VLSI is managing the complexity of systems which will soon contain millions of components [69].

In response to this challenge many research programs have been started and VLSI continues to be a popular topic for researchers from many different disciplines, in both industry and academia.

The next section outlines work related to the tool presented in Chapter 5, and gives the background for further applications of the Wright system, as discussed in Chapter 6. Section 2.3 reviews work on programming environments, and the final section discusses VLSI programming environments.

2.2 VLSI Design Tools

2.2.1 Introduction

The demand for VLSI implementations of systems has caused the development of a wide range of tools and techniques, many of which have been borrowed from other design disciplines. In reviewing some of this work I am particularly interested in two aspects relevant to this thesis:

- design notations
- tool integration

The design tools which deal with physical layout are given closer examination, since this is the problem area addressed by Stick-Wright in Chapter 5.

2.2.2 Structured Design

As in many other design disciplines (notably computer programming), the need to control design complexity has led to the formulation of sets of design principles. Buchanan [9] gives the following principles for *structured IC design*:

Modularity allows work to be partitioned into manageable sub-goals, also permitting work to be distributed among more than one designer. Modules can be re-used in later projects.

Hierarchy i.e. the vertical partitioning of designs; modules are themselves composed of smaller sub-modules. There are also hierarchies of abstractions (e.g. geometry, devices, circuits, logic, floorplans), allowing information hiding at each successive layer (thus making descriptions smaller and permitting efficient algorithms that exploit the hierarchy).

Regularity Regular structures are simpler to describe, manipulate and understand. Regularity in the physical domain (rectangular or hexagonal blocks designed to abut together) can yield high packing density. Irregular layout can result in great amounts of wiring (*spaghetti layout*).

Locality Modules can only be accessed through well defined external interfaces, similarly, the internals of a module can not depend on external factors, except as described through the interface. This *black box* approach allows *proof by construction* techniques to be performed (e.g. hierarchical design rule checking (DRC)).

Parameterisation (also Programmability). Circuit structures (e.g. RAMS, ROMS, datapaths) can be described algorithmically with particular instances being created using some personality matrix or set of parameters.

These are all familiar concepts from computer programming, Buchanan shows their application to hardware systems, with reference to their analogies in *structured programming.* An early appraisal of hierarchical design by vanCleemput [90] draws attention to some possible problems; hardware systems can be viewed as having several different hierarchies (Buchanan identifies behavioural, structural and physical hierarchies) and each class of hierarchy can have many realisations. The multiplicity of hierarchies is seen as problematic because of the expense of maintaining correct mappings between them. Buchanan addresses this problem by employing the co-ordinode, a data structure which unifies the hierarchies with a common representation. The co-ordinode together with associated wire and transistor models are implemented using the object oriented language SIMULA [6]. The object oriented programming paradigm has proven to be popular among some VLSI researchers, a major reason being that it fully supports the principles of structured design. Object-oriented systems are discussed more fully in the programming environment section.

Cardelli [12] uses algebraic techniques for describing VLSI systems at various levels of abstraction. Circuits are described using net-algebras and he shows

how a behavioural description, the Clocked Transition Algebra, can be translated through intermediate descriptions (Connector Switch Attenuator - planar sticks - grids) down to layout. A practical demonstration of this design style, the layout editor Sticks&Stones [13] , uses the data abstraction mechanisms of the functional programming language ML [25] to implement the net-algebra representation. The major feature of this representation is that circuit structures can only be manipulated by their external named ports (thus applying the principles of modularity and locality). Picture composition is achieved by linking port names (not by stating geometrical position or displacement). This saves the user from having to deduce translations, mirrorings and rotations which enable the construction of the picture. By exploiting the referential transparency of the host functional language (an example of the application of the principle of locality), *correctness by construction* proofs can be performed (e.g. hierarchical design rule checking [94]). The actual design notation is described in the next section.

The Palladio project [27] also exploits the power of abstraction in its technique of *multiple perspectives.* Circuit design is viewed as a process of incremental refinement of structural and behavioural specifications over a range of perspectives (circuit level to PMS). Different programming paradigms are applied to the various aspects of the design process; a rule based logical language is used in behavioural specification and simulation, while an object oriented approach is used in structural specification. The Palladio system is based on the following premises about circuit design:

- it is a process of incremental refinement.
- it is an *exploratory* process in which design specifications and design goals co-evolve.
- circuit designers need an integrated design environment (i.e. compatibility between a range of tools).

These premises are compatible with the Wright system's design goals, and I will later argue the importance of joint textual/graphical notations for design *exploration.*

The application of algebraic techniques (e.g. formal semantics) and the use of structured design are widely reported in the computer science literature for controlling the the complexity of computer programs and also for guiding the design of programming languages [86]. The previous projects have demonstrated the usefulness of 'programming techniques' to hardware design, and the need for the abstraction mechanisms they provide. In later sections the importance of pictorial notations will also emerge. The view of hardware design as a programming exercise, and consequently accruing benefit from these techniques, is becoming more prevalent and is even making an impact on some industrial practitioners.

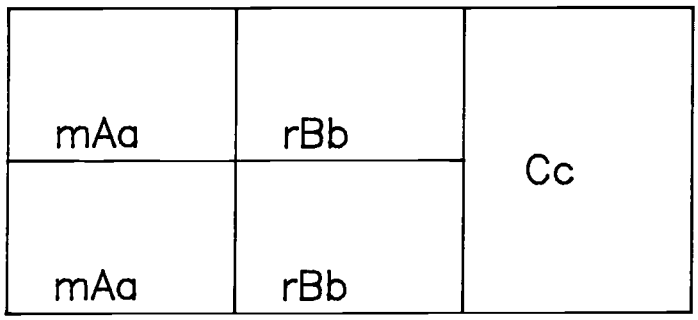
2.2.3 Design at the Mask Data Level

The design tools surveyed in this section all deal directly, at some stage, with mask geometry. This approach gives the designer precise control over the physical realisation of a circuit, but with this freedom the potential for error is introduced.

The embedded design language [49] was the earliest attempt at using high level programming techniques for circuit design. Such systems are simple to develop and can exploit the power and familiarity of their host high level language (e.g. ILAP [30] uses IMP [73] and Sticks&Stones [13] uses ML [25]). These systems can be viewed as being too powerful in the sense that it is difficult to prove properties of large software systems (we could not easily prove that an ILAP pad-placement program will always give a correct result). The systems are weak in the sense that since they do not directly support two-dimensional programming (i.e. layout generation), the specification of circuit constructs can be obscure and inflexible. Major benefits of these systems include ease of implementation (no new compilation technology is required) and ease of use (i.e. assuming that the host language is already known and is itself easy to use). Much, of course,

depends on the quality of the embedding. In the following discussion I will mostly be concerned with how various systems deal with cell composition, leaf cell design is usually carried out by graphical editors that generate the embedded language as output.

The Lubrick cell assembler [80] uses an embedded language (Pascal is the host language). It is a cell composition system which assumes the correctness of leaf cells. This assumption allows the inclusion of arbitrary geometry in leaf cells, although cell boundaries must be rectangular. Lubrick uses the concept of type-directed connections to facilitate the joining of cells. Essentially this involves inserting an appropriate routing channel between cell boundaries, which depends on the type information contained in the port definitions. Composition of cells is specified using Pascal functions. The construction of the following structure is illustrated:

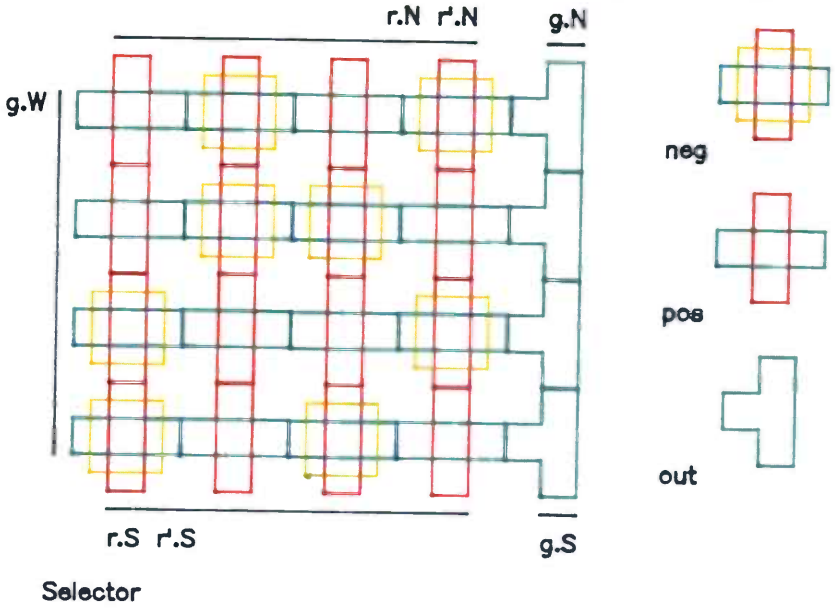


where cells `Aa`, `Bb`, and `Cc` are leaf cells, `mAa` is the mirror image of `Aa` in the Y-axis, `rBb` is the rotation of `Bb`. The corresponding LUBRICK code is:

```
p:=OPENCELL('EXAMPLE')
    p1:=REPY( SYMY( GETCELL('Aa') ),2 );
    p2:=REPY( ROTP( GETCELL('Bb') ),2 );
    p3:=GETCELL('Cc');
p:=CLOSECELL(RIGHT(RIGHT(RIGHT(p,p1,1,0),p2,1,0),p3,1,0));
```

This would look like the following in ILAP notation:

```
SYMBOL("EXAMPLE")
  DRAWMX("Aa",SX("Aa"),SY("Aa")*i) %for i = 0,1,1
```

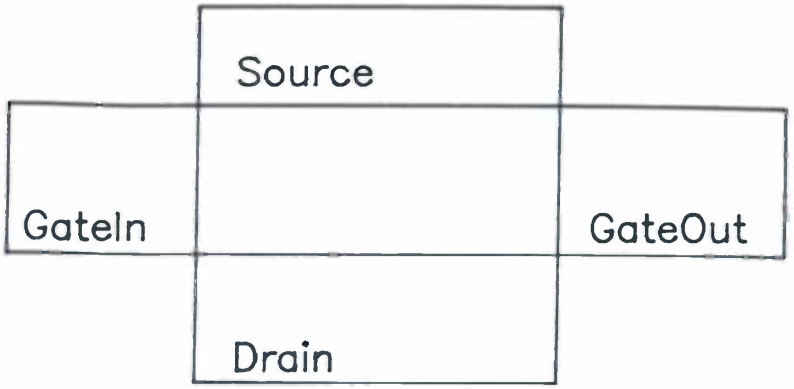



```
let sel n =
  for i in 1::exp(2,n)
  iter (for j in n::1
        iter bit(i-1,j-1)=0 =>
               pos [: g.E -- g.W :] (neg{r.?\r'.?}) |
               pos [: g.E -- g.W :] (pos{r.?\r'.?})
        with [: g.E -- g.W :])
        [: g.E -- g.W :] out
  with [: r.S -- r.N ; r'.S -- r'.N; g.S -- g.N :]

whererec bit(i,j) =
  j=0 => i mod 2 | bit(i//2,j-1);
```

Figure 2–1: A Sticks&Stones Selector Circuit

component relations are derived from the design rules and the composition of cells. All necessary relationships are discovered by the system, but the user can manually specify relationships using the *constraint* mechanism. With this feature the user can control circuit features directly (e.g. impose maximum lengths on wires and specify transistor sizes). Figure 2–2 is an example of the Plates language, namely the definition of the circuit primitive `PassTransistor`.

Figure 2–2 is a small example of the Plates language, a description of a shift-register is 45 lines long. Part of the reason for the verbosity of this notation is that the overlapping of boxes is a very low level abstraction, and is used in both the definition of primitives (as above) and in the composition of cells.




```
Primitive  PassTransistor
             (GateIn {Position (Left), Access (Top, Bot, Left) },
              GateOut {Position (Right), Access (Top, Bot, Right)},
              Source {Position (Top), Access (Top, Left, Right) },
              Drain {Position (Bot), Access (Bot, Left, Right)});
  Begin
    Instantiate D using Diffusion,
                 P using Poly with (Left outside D.Left,
                                    Right outside D.Right,
                                    Top inside D.Top,
                                    Bot inside D.Bot);
    DefinePorts GateIn is (P outside D.Left),
                GateOut is (P outside D.Right),
                Source is (D outside P.Top),
                Drain is (D outside P.Bot);

  End ! PassTransistor !
```

Figure 2–2: A Plates Primitive

While allowing the specification of very general constraints, this is bought at the expense of unwieldy descriptions. The goal of having a metric-free representation is, however, a useful contribution and is developed further in the ALI system [48].

ALI is an example of an extended language (Pascal + circuit description syntax and semantics). The ALI system bases its representation of circuit structures on a set of linear inequalities. These inequalities refer to the relative placement of objects within the circuit, either as supplied by the user or derived from the design rule file. This approach liberates the 'programmer' (in the ALI project circuit design is viewed as programming) from specifying sizes and positions. Cell

composition is achieved by *stretching* the cells, i.e. constraints are determined which cause ports to match up exactly. The design rules can provide many of the required object relations (e.g. wire spacing). The ALI system must check that all relations between circuit entities are specified. This condition, known as completeness, involves computing the transitive closure of a graph. Although this has $O(n^3)$ time complexity, n is never more than the number of objects in one cell. The execution time due to the solution of the set of linear inequalities is proportional to the number of inequalities. Figure 2–3 is an example ALI program (with corresponding picture).

```
Chip simple;
  const
    hnumber = 10;
    length = 20;
    width = 6;
  boxtype
    htype = array [1..hnumber] of metal;
  var
   i:integer;
  box
    horizontal:htype;
    vertical:metal;
  begin
    for i:=1 to hnumber - 1 do begin
      above(horizontal[i],horizontal[i+1]);
      glueright(horizontal[i],vertical);
      xmore(horizontal[i],length)
    end;
    glueright(horizontal[hnumber],vertical);
    xmore(horizontal[hnumber],length);
    xmore(vertical, width)
  end.
```

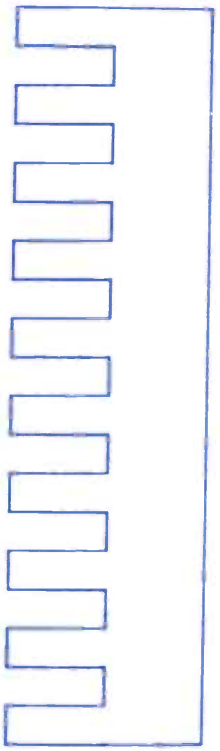


Figure 2–3: An ALI program

Although the ALI team claim that their system's avoidance of explicit sizes and positions is novel with respect to other systems, including graphical editors, this is not true of the various symbolic layout systems. The placement of devices on a virtual-grid (described in the next section) is no more concrete than the gluing, separation and overlap primitives of ALI. While these are in-

teresting notational devices (an abstraction made use of in the text formatting language TEX) they are not any more or less descriptive than a symbolic layout stick-diagram or program script. Unlike the symbolic layout tools, ALI doesn't completely remove the possibility of a design containing an explicit size (as in Figure 2–3), and does not directly deal with layout at the level of abstraction corresponding to wires and transistors. ALI does provide an implicit DRC and circuit extraction capability (an improvement over the previous systems), but both these functions are only provided on a completely instantiated design (i.e. they are batch oriented).

The Magic [67] design system supports incremental design by representing mask data in a data structure called *corner stitching.* This representation supports a design style called *logs* which allows the manipulation of abstract layers (which represent transistors, wires and contacts) in a manner similar to symbolic layout systems. In Magic, however, logs appear with actual sizes and positions. Corner stitching supports incremental design rule checking and a wire perturbation action called *plowing.* The major advantages over the previously described systems are:

improved interaction The design representation is graphical, all editing is done through a graphical interface, making use of command menus and providing multiple windows onto the design. The algorithms associated with the editor's major operations (DRC, routing and plowing) are incremental (i.e. they avoid re-evaluating the whole design, thus speeding up response times).

higher level of abstraction Although Magic allows the direct manipulation of mask geometry, the use of abstract layers means that the system has knowledge of the circuit's structure (i.e. not just its physical layout), thus allowing trivial circuit extraction for verification and simulation.

Whitney [93] describes the hierarchical composition of cells built using the Pooh design representation. In Pooh transistors and wires are represented as line segments which have associated widths and separations. The geometry in a

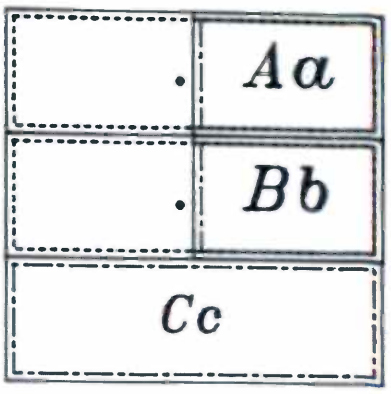
Pooh design can form arbitrary angles. When cells are composed (by matching a pair of sides) only information about the new external boundary is retained. A further development of the Pooh system [84] adds the restriction that geometry can only take 90° and 45° angles. By doing this the system removes the need for real arithmetic thus greatly improving the efficiency of the system with little loss in generality (arbitrary angles do not give much greater layout density than 90° and 45°). Pooh is actually much more than a layout language, its wire/transistor model form the basis of a large design system which also incorporates a symbolic layout interface.

The major benefit in dealing directly with mask data is that the physical result can always be determined (usually without much delay) and is directly controllable. A major disadvantage is the lack of freedom the designer is afforded with regard to cell stretching and port juxtaposition, (the ALI system's cell stretching ability is bought at the expense of limited layout control and interaction, a conscious design decision). The last two systems (Magic and Pooh) guard against the possibility of incorrect layout by integrating design rule checking into the design entry process.

2.2.4 Symbolic Layout

Symbolic Layout systems allow the specification of integrated circuits as sticks, i.e. diagrams which show the topology of a circuit without showing the precise physical placement and sizing of the circuit components. The designer provides the relative position and connectivity of a circuit components, from which the system deduces the final transistor and wire geometries. Sticks can be described textually as well as graphically, and also as hierarchies of sticks. Layout is produced from stick descriptions by the action of a compaction algorithm. The tool described in Chapter 5, Stick-Wright, is a symbolic layout editor, making use of both stick diagrams and a textual notation. Here is an example stick diagram and textual description (a quick preview of Stick-Wright, in fact), using the example introduced in the last section:

```
Cell Example (;;;) =
  Abut
    <Aa~[2]> <Bb^[2]> Cc.
```



where < .. > indicate vertical composition, [...] iteration, ~ mirroring in the Y-Axis and ^ clockwise rotation.

The earliest symbolic layout systems used a technique known as *fixed-grid* [43]. Here components are arranged on a grid which is spaced out according to the worst-case design rule (i.e. the largest of the minimum distances that any of the rows or columns has to be away from its neighbour for the circuit to operate correctly). The grid can not be compacted further.

The next generation of tools, the *relative grid* systems use a grid initially to specify the placement and electrical connectivity of the components, but a compaction process could then attempt to improve the layout density by moving components off the grid lines. The Williams [95] relative-grid system used stick-diagrams to enter the design (he uses the term *gridless*). Williams describes the compaction process as graphical compilation. The compactor in this system treats placement separately in the X and Y directions. In Slim [18] Dunlop makes an improvement to the Williams compaction strategy by adding a critical path heuristic to guide trade-offs between X and Y compaction. The Slim system also uses partitioning techniques to reduce the number of tolerance tests on large layouts.

A combination of the fixed and relative-grid compaction schemes, *virtual-grid*, was introduced in the MULGA system [92]. Virtual-grid components can move about during compaction, but must stay on their grid lines, and the distance between grid lines can also vary. The textual stick language of MULGA, ICDL, evolved into the language ABCD [75] in the VIVID system. VIVID uses a

hierarchical virtual-grid compactor which only compacts each cell once. The compactor can extend the cell boundaries of a previously compacted cell to conform to a new placement context.

In ASTRA [72] compaction is linked with the system's floorplanner. The compactor is called initially to determine the minimum size for a given cell. This information is then used to determine cell placement. The compactor is called again with a set of port constraints in a final cell assembly stage. This is illustrative of the oscillations which can occur between bottom-up (compaction) and top-down (floorplanning) design activities.

In the Hill system [46] the compaction scheme (based on Williams) is enhanced by topological flexibility; if during compaction it can perturb the circuit's topology while maintaining the original circuit structure, it does so (if this saves area). The Hill system proposes a design language which allows flexibility in the specification of a cell's external appearance, its *template*. The system then checks that the given implementation for that cell can be distorted to meet the given context. This flexibility is specified by giving a partial order for the ports in either the X or Y direction (the distortion mechanism only copes with freedom in one dimension). At each level in the hierarchy three layout modes are available; composition, placement and graphical. Composition and placement are both text descriptions (placement is a more verbose version of composition modes, where every cell instantiation has a unique name and it provides better checking and simulation facilities), while the graphical interface is a stick editor. Figure 2–4 is an example of composition mode.

In Figure 2–4 `basic` is a previously declared cell. This definition of `chain` is recursive, the Hill implementors observe that in the same way as iteration is good for describing array structures, recursion is descriptive of tree structures (e.g. hierarchical multipliers). There is no graphical *dual* for the above description, except for that of a particular instantiation (they show a picture of `chain(3)`). What the Hill system does not provide is an integrated environment which allows switching between the different design notations, the system they describe is

```
Cell chain (n:int);
Temp Pins over: cbits;
          cin: poly;
          data: Array [0..2^^n -1] Of
                Record in: cbits; out: poly End;
          ps1, psr: Array [1..n] Of ps;
          Order Implicit:Ver;
                Top: over, cin;
                Bottom: data Reverse;
                Left: ps1;
                Right: psr;
          Constraints
          Begin
            cin Above data[0].out;
            over.p LeftOf data[2^^(n+1)].out
          End
Pmet
Composed
Begin If n >= 2 Then
        chain := basic(n) Ver
                 (chain(n-1) Hor Chain(n-1))
      Else
        chain := basic(n)
```

Figure 2–4: An example of Hill composition mode

batch oriented. The graphical representation of programming language features which avoid full instantiation is discussed in Chapter 5.

Since symbolic layouts are a level of abstraction above layout, they are buffered from design rule variations. Bergmann [5] also shows that stick descriptions can give a degree of technology independence within a process family. As a final example of VLSI design notations Figure 2–5 is the last section's example written in Bergmann's idiom description language Virgil.

```
Composition Cell Example
  AA = ^^ A INY ^^ A INY
  BB = ^^ B @1 ^^ B @1
  Example = >> A >> B >> C
End
```

Figure 2–5: A Virgil Composition Cell

Virgil is a cell composition language where the leaf cells are described using a virtual grid. Figure 2–6 shows the textual form of such a cell.

```
Leaf Cell Side = ( 0, 0, 1, 2)
  gnd.e: mport @ ( 1, 0 )
  gnd.s: mport @ ( 0, 0 )
  gnd.n: mport @ ( 0, 2 )
  mwire @ gnd.e -> gnd.a -> gnd.n
  in.e: pport @ ( 1, 1 )
  in.w: pport @ ( 0, 1 )
  pwire @ in.e -> in.w
End
```

Figure 2–6: A Virgil Leaf Cell (text)

Virgil leaf cells are usually created using the graphical representation provided by a graphical editor (see Figure 2–7).

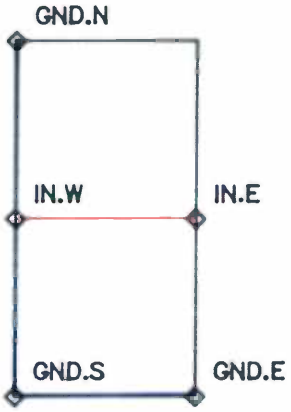


Figure 2–7: A Virgil Leaf Cell (graphics)

Symbolic layout will not suit circuit designers who are used to optimising every last box in a layout by hand, and who want to create unusual geometries and perform 'neat tricks'. The claims of the symbolic lobby of process and technology independence will not worry a designer who has a stable design facility. However, some of the arguments about symbolic versus hand layout sound very much like the arguments you still here today about assembly code versus high level languages. Given the complexity demands of large VLSI systems and the increasing quality of symbolic layout compaction schemes, it seems reasonable

to expect symbolic layout to continue to gain ground, and it is the chosen abstraction for the design tool in Chapter 5. A more detailed survey of symbolic layout is given by Newton [62] and a review of commercially available systems is given by Taylor [83].

2.2.5 Silicon Compilers

Silicon compilation is a rather overloaded term, frequently to be found in marketing hyperbole, but is nevertheless an extremely apt description of those tools which automatically synthesise complete IC layouts from some input specification. The term has been applied to systems using structural design descriptions (e.g. Lattice Logic's gate array system [26] [44]) and also more ambitious systems which process behavioural descriptions (e.g. the algorithmic LISP like design language of MacPitts [81] [82]). In systems like MacPitts the compiler is not only responsible for the placement and routing of the circuit primitives implicit in the design language (called *organelles* in MacPitts), but it must also synthesise the control logic which governs the operation of the circuit.

While the task of finding efficient layouts for arbitrarily complex structural descriptions is no easy task, the synthesis of layout from behavioural specifications also involves much more complex partitioning and architectural decisions. So far the most successful silicon compilers have made this task more feasible by restricting both the problem domain and the target architecture: MacPitts has a fixed datapath/controller architecture, and the primitive operators in the design language have directly corresponding hardware implementations.

A further example of this 'target application' approach is the FIRST [16] silicon compiler which transforms a high-level design language description of digital signal processors into IC layout. The internal architecture is restricted to hard-wired networks of pipelined bit-serial operators. The FIRST system supports functional simulation of the design language and can give precise performance estimates. This combination of high-level design language, functional simulation and performance prediction together with a fully automated layout synthesis

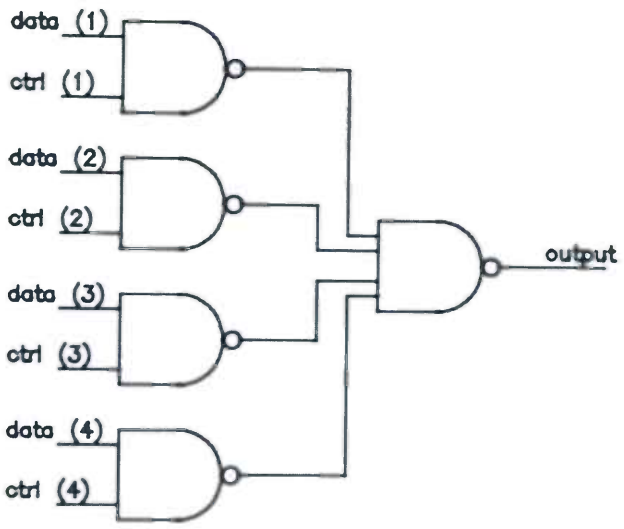
stage enables FIRST to provide silicon implementations for designers with no previous IC design experience, i.e. systems designers can produce chips. By fully automating the layout synthesis stage, the layout can be guaranteed to be free of human introduced layout errors.

So far the tools in this section have not made much use of graphical interfaces, however, Lattice Logic's gate-array compiler is a notable exception. As a commercial product it has had to meet the demands of design engineers who require schematic entry (in preference to the structural design language MODEL). For this reason a schematic entry system was implemented which converts circuit diagrams into MODEL language constructs. A problem encountered during the implementation, and consequent maintenance of this system [8] was that the MODEL language was also undergoing design iterations. Since the implementation of the picture → language translator was not directly connected or driven by the language's specification (i.e. grammar) the updates needed significant hand re-coding. A further problem with schematics entry for the MODEL language is that schematics, at least in their current form, can not exploit the *structured programming* features of MODEL, i.e. parameterisation, iteration and conditionals, and so the MODEL generated by the schematics package is in a fully instantiated form. Figure 2–8 shows the schematic and corresponding MODEL for a four-way multiplexor, and Figure 2–9 shows a parameterised version of the multiplexor.

These problems, and the application of the Wright system to silicon compilation in general, are discussed in the final chapter.

2.2.6 Floorplanning

Floorplanning in the context of VLSI design is the act of determining a relative placement of system elements so as to minimise the communication (wiring) costs between them, and hence the overall size. Additional aspect-ratio constraints may also be specified. Floorplanning is used in the initial stages of design to place major circuit elements. Although the exact size of the elements to be




```
PART mux[data(1:4),ctrl(1:4)] -> output
  SIGNAL temp(1:4)
  nand[data(1),ctrl(1)] -> temp(1)
  nand[data(2),ctrl(2)] -> temp(2)
  nand[data(3),ctrl(3)] -> temp(3)
  nand[data(4),ctrl(4)] -> temp(4)
  nand[temp(1:4)] -> output
END {of mux}
```

Figure 2–8: A MODEL four-way multiplexor

```
PART varimux (m) [input(1:m),control(1:m)] -> output
  SIGNAL temp(1:m)
  INTEGER i
  FOR i = 1:m CYCLE
    nand[input(i),control(i)] -> temp(i)
  REPEAT
  nand[temp] -> output
END {of varimux}
```

Figure 2–9: A MODEL parameterised multiplexor

placed cannot be found (except by full implementation), estimates can be made either by expert knowledge or by statistical methods [3].

Heller et al. [28] [51] use graph dualisation to determine a chip floorplan from a connectivity graph. The graphic input is first tested for planarity and

then dualised. Planarity can always be enforced by the introduction of nodes corresponding to routing channels. The rectangular dual which best meets these constraints is picked. Unfortunately, this process has exponential time complexity.

Brebner and Buchanan [7] add a textual interface to the Heller et al. algorithm and automatically produces a suitable graph for dualisation. Because the algorithm is applied to a hierarchical structure, with small graphs at each level, the complexity issue is partly avoided.

Otten [66] describes the application of planar projection to floorplanning. This involves embedding the interconnection graph with n nodes into (n-1) dimensional space. Projections can then be made onto a 2-dimensional plane. The edges of the graph are sized according to the degree of connectivity between the nodes; short distance implies high communication. A number of projections are made and the one which least disrupts the inter-node distances is chosen. The last stage, called slicing, is to partition the 2-D plane into rectangular areas determined by the projection.

An experimental environment using the Heller and Otten methods is described by Schmid in [79]. The Otten algorithm is described as being fast enough for an interactive system, but solutions meeting all communication and size criteria cannot always be found. The system uses the Otten algorithm to help the user refine a specification that will drive the Heller algorithm.

The emphasis in these tools is design automation, rather than computer aided design. Because of the computational expense involved in automatic floorplan tools, it seems reasonable to expect graphical design aids will continue to be used for some time, and hence the techniques described later in this thesis can usefully be brought to bear on this subject. Indeed, it is later argued that graphical design aids are crucial to the effective use of silicon.

2.2.7 Verification

There are being developed verification tools which attempt to prove that implementations of systems conform to their specification. This is an extremely important objective as exhaustive simulations are already prohibitive in even modest sized systems.

Milne describes a calculus for circuit descriptions called Circal [58]. He demonstrates an application of Circal by verifying the correctness of a simple silicon compiler [59]. In this example Milne describes a semantic function which maps Nor expressions (the input language of the compiler) to Circal and a function which maps layout (the output of the compiler) to Circal. By showing that both Circal expressions are equivalent, in this example they turn out to be the same, the correctness of all possible circuits of the system is proven. In more complex examples it is predicted that the expressions would have to be manipulated using Circal's laws to prove the equivalence. Traub [87] describes an experimental Lisp system which provides machine assistance in manipulating Circal expressions. Circal has also been used as a basis for an interactive simulation system, where Circal expressions are *animated* on a graphics screen

Barrow [4] presents a system, Verify, which performs functional verification of circuits, without regard to timing issues. The system attempts to prove that the description of the behaviour of the circuit at one level in the hierarchy matches the behaviour derived from the interconnection of its constituents. The proof is made automatically as far as is possible, using a Prolog based algorithm, and has an interactive mode where the user can guide the proof (i.e. suggest application of laws).

The work in this thesis is relevant to design verification in two areas, expression manipulation (including proof-editing), and implementing interactive simulation systems.

2.3 Programming Environments

A *programming environment* is an integration of the tools associated with program development (text editors, compilers, interpreters, debuggers, display programs, etc.) within a unifying framework. A component of most programming environments is a *structured editor* (sometimes also known as a *structure editor*, reflecting a subtle change in emphasis). This tool combines the functions of a text editor with that of a parser to yield an editing system which has knowledge of the structure being edited, and can hence forbid illegal constructs and permit actions which make use of the structure (e.g. movement and deletion of large sections of program). The Mentor project [17] built a structured editor for Pascal which provided a tree manipulation language called Mentol. Using Mentol it is possible to specify complex tree traversals, insertions and deletions. The following piece of pascal:

```
if X>0 then P(X,A[Y,Z])
else begin
     Y:=Y*2;
     X:=0
     end
```

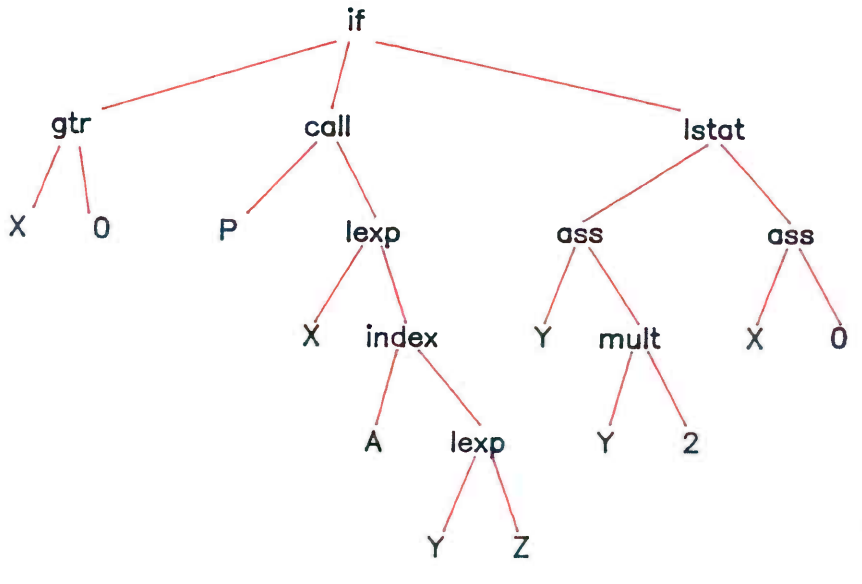


Figure 2–10: A Mentol abstract syntax tree

is represented by the abstract syntax tree in Figure 2–10, and is transformed by the following Mentol commands:

```
:@TOP                  % place current marker at top of tree
S2 X S3                % exchange sons two and three
S2 S1 I S3             % insert son3 at son2 of son1
S3 C &                 % replace S3 with ...
Z:=0                   % the parse tree of this line
```

into the tree having the following unparsing:

```
if X>0 then
     begin
     Y:=Y*2;
     P(X,A[Y,Z]);
     X:=0
     end else Z:=0
```

The Cornell Program Synthesiser [85] introduces the terms *template*, *placeholder* and *phrase*. A template is a predefined formatted pattern of lexical tokens and *placeholders*. A placeholder identifies locations in a template where insertions can be made. A phrase is a sequence of lexical tokens (e.g. keywords, numerals and identifiers). Each placeholder designates the syntactic class of permissible insertions. An example of a template (from [85]) is:

IF (*condition*)
THEN *statement*
ELSE *statement*

where *condition* and *statement* are placeholders. Programs can be entered by expansion of templates either by the insertion of further templates into placeholders (a top-down approach) or by entering a phrase at a placeholder (a bottom-up approach). The bottom-up entry method requires the use of a parser to prevent illegal sentences being inserted.

Many other programming environments include editors which have features in common with the Cornell Program Synthesizer, including the editor module of the Wright system (Chapter 3). I shall continue to use their terminology.

An all embracing approach to programming environments can be seen in the Smalltalk project [24] where everything from the virtual-machine architecture to the high-level programming paradigm has been conceived as a whole. The object-oriented approach taken in Smalltalk lends itself to tool integration, making possible natural descriptions for inter-tool communication and resource sharing. To describe the essence of Smalltalk it is necessary to introduce some of its vocabulary:

object A component of the Smalltalk system represented by some private memory and a set of operations.

message A request for an object to carry out one of its operations.

class A description of a group of similar objects.

instance One of the objects described by a class.

method A description of how to perform one of an object's operations.

In Smalltalk, computations are described by the interaction of objects, which communicate using messages. When an object receives a message it applies the corresponding method to perform the desired computation. Objects with shared properties are structured in classes, and new classes can be derived from existing classes using the subclass mechanism. Subclasses inherit all of the properties of their superclass, however, they can modify these properties and add their own. For example if the class 'WindowManager' has a method 'DrawFrame' which draws a thick border round a window, to provide a new window manager which draws only thin borders round windows, would simply involve creating a subclass of 'WindowManager' which overrides the method 'DrawFrame' with one that draws thin borders. The concepts of inheritance and message passing can lead to very elegant descriptions and implementations of systems.

Similar developments have led to the SYMBOLICS 3600 LISP machine which has been used to develop a VLSI CAD system NS [15]. The authors attribute the

success of their system to abstraction mechanisms provided by the LISP machine; the Flavors object-oriented programming language, large uniform virtual address space and procedure-data duality (procedures are *first-class citizens*, i.e. they have a well-defined data representation which allows them to be manipulated just like any other data object). Flavors makes an alteration to the Smalltalk and Simula class structure by providing the mechanism of *multiple inheritance.* Flavors can inherit methods from multiple-superclasses, i.e. new flavors can be constructed by mixing existing flavors.

An important aspect of programming environments is the level of support given to graphics. Both the Symbolics and Smalltalk systems make extensive use of the graphics capabilities present on their workstations, indeed, this is a measure of the success of their programming paradigm.

Programming environments have been created for traditional procedural languages (e.g. Pascal) and developments have been made in not only syntax-directed editing but also in such things as debugging aids, source code version control systems and incremental compilers. Research programs in this area include Gandalf [57], POE [21] and SAGA [11].

In the area of syntax-directed editing a popular formalism to specify context-dependent language features has been the attribute grammar [39], and this is the formalism that forms the basis of the Wright system. Major work in this area has been done in the development of the successor to the Cornell Program Synthesizer, namely, the Synthesizer Generator [71]. This project has successfully developed optimal-time incremental attribute evaluators and has developed methods for reducing the storage requirements of attributes. The Cornell system has been used in many applications, including the following:

- Pascal editor with full static-semantics checking.
- An editor for partial-correctness program proofs using Hoare-style logic.
- A full-screen desk calculator.

- a text formatter.

- a mathematical equations formatter.

Further discussion of the Cornell project is made in the next chapter, following the introduction of the formalism of attribute grammars.

2.4 VLSI Programming Environments

2.4.1 Introduction

The work in this thesis is primarily motivated by an interest in applying the techniques of programming environments to VLSI design. This area has seen some interesting research projects, of which SAM [88] is perhaps the most relevant to this thesis. In SAM, Trimberger implements an embedded language package using the SMALLTALK programming environment. It contains the following key features:

Text and graphics The user can view the design as either text or graphics.

One internal representation The text and graphics windows onto a design are different views of the same internal data structure. Hence a change to this structure causes the regeneration of both views.

Trimbereger identifies two areas of difficulty in reconciling textual and graphical representations:

expressions what happens when an x coordinate described by the expression $3*w+4$ is transformed by a graphical command to a new position ? Say the value of the expression was 10 and is now 13. The following substitutions are possible in the text version:

13	destroy parameterisation
$3 * w + 7$	add a constant (translate)
$(13/10) * (3 * w + 4)$	multiply by a constant (scale)
$3 * w + 4$ where $w = \mathbf{3}$	change the value of the identifier

iteration If a change is made graphically to one instance of a cell invoked by a loop statement, should only it, or all the instances change ?

These problems are partly due to the level of abstraction used in the system, i.e. the use of absolute values, and also the nature of the graphics commands available. The approach taken in the Wright system to solve these problems is presented in chapters four and five.

There are now commercial products emerging which are showing a high level of tool-integration, and which make use of both textual and graphical interfaces. Representative of these are SDA's SKILL system [45] and SDL's GDT (generator development tools) [10]. Both these products have procedural design languages for the development of module generators, and allow these descriptions to be entered using graphical editors. Neither of these sytems, however, supports graphical and textual manipulation of the same object at the same time. Central to both these systems is the design data-base which is closely tied to the respective design language (SKILL and L). All the tools involved in the design process (simulators, routers, compactors, schematic editors etc.) communicate through this shared design representation.

The application of attribute grammar techniques to VLSI design systems has also recently been suggested by Jones and Simon [36]. Their work has been based very closely on the Cornell Synthesizer Generator, and has so far focused on the development of evaluators for *circular* attribute grammars. The need for circular attribute grammars has arisen since they are interested in describing some of the more dynamic features of circuits which are inherently circular (e.g. delay propagation and logic simulation). The Wright system is primarily concerned with the relationship of programs to pictures, and has only needed to make use of *non-circular* attribute grammars.

2.4.2 Interactive ILAP

Interactive ILAP was my final year project as an undergraduate [54]. I include a brief description of it here because the work reported in this thesis has largely followed on from it, and it amply illustrates some of the problems I am addressing.

As introduced before, ILAP is VLSI layout language embedded in the programming language IMP. Figure 2–11 illustrates the ILAP design cycle.

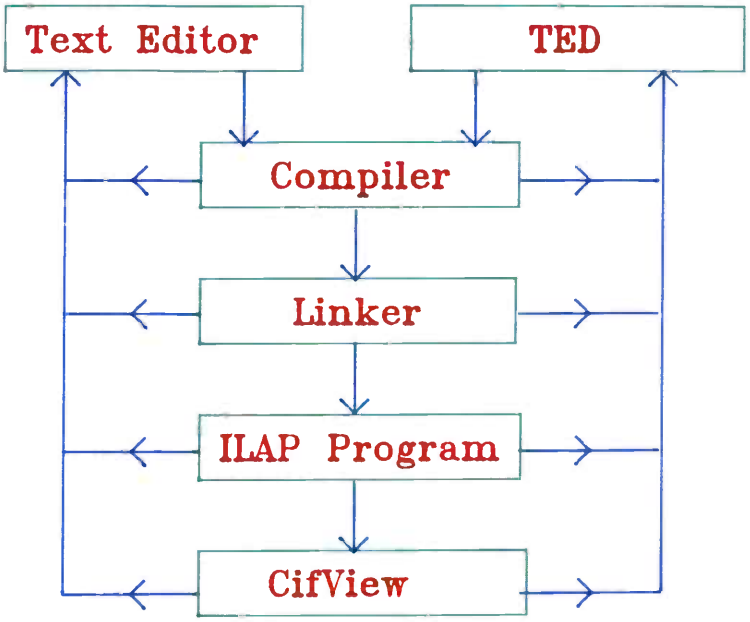


Figure 2–11: ILAP Design Cycle

During the evolution of a circuit the designer makes use of the following programs:

text editor In the university there are an abundance of full-screen text editors available, some of which, e.g. EMACS, can be instructed how to format language constructs.

TED Leaf cells (i.e. cells containing only geometry primitives) can be constructed using the Ted graphical layout editor.

compiler The ILAP package makes use of the IMP compiler, which even on a lightly loaded or personal machine can take a few minutes to compile a moderately sized design.

linker Resolves external references in the object code generated by the compiler providing an executable image (this stage is eliminated in systems with dynamic linking).

ILAP program The compiled ILAP program is now executed to generate the CIF file [56] corresponding to the design.

CIFview Finally the user can observe on a display (or plot) the geometry that has been constructed.

All the tools in the design cycle view the design as a whole; if the coordinates of a cell translation in a design were entered in reverse, in order to correct that one point all of the above tools would have to be re-invoked and the whole design re-evaluated. Interactive ILAP's brief was to remove the redundant re-evaluation thus reducing design time. In order to do this a programming environment approach was taken and some of the features of the systems described by Trimberger [88] and Medina-Mora [57] were incorporated.

Figure 2–12 show the major components of Interactive ILAP.

editor Interactive ILAP provides a syntax-directed editor for IMP. The editor also provides the control mechanisms through which the user directs the execution and display of the system. The editor maintains two data structures; a list of analysis records (collapsed parse trees) associated with each source line, and an array of pseudo-machine code generated from the analysis records.

interpreter The interpreter executes the pseudo-machine code maintained by the editor. The interface between editor and interpreter allows direct control of execution and comprehensive monitoring facilities.

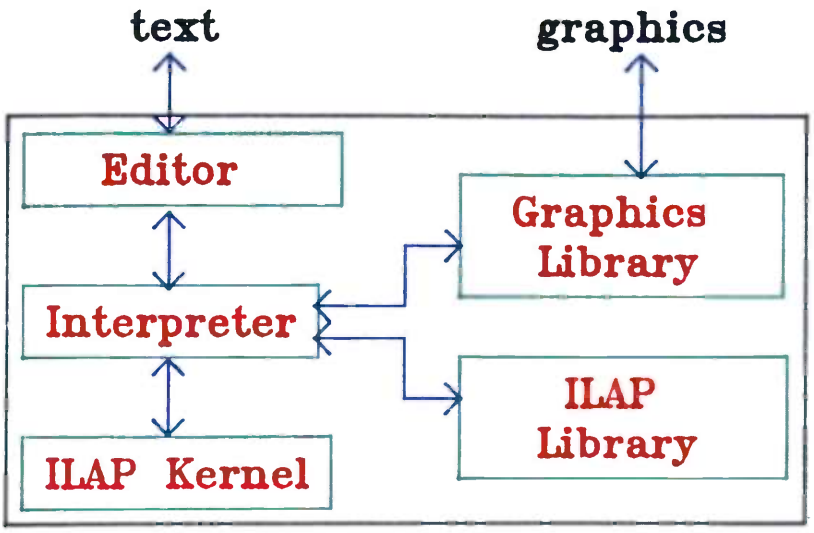


Figure 2–12: Interactive ILAP Structure

ILAP kernel The normal ILAP kernel produces CIF layout as output. Interactive ILAP has its own kernel which generates internal graphics code at run time. This dynamically generated code can then itself be executed causing calls on graphics procedures and, hence, pictures.

Graphics library Interactive ILAP uses EDWIN [29] for the presentation of circuit layouts. EDWIN is particularly useful because of its portability over a range of hosts and graphics devices.

ILAP library The ILAP library contains programmable structures built on top of the ILAP kernel. Since these are independent from the kernel and appear as external references, Interactive ILAP can use them without modification.

The textual representation of an ILAP design, i.e. the IMP program, is displayed and edited using a normal VDU. The editing functions provided include the normal structured editor tree traversal and manipulation procedures, and source lines can be entered bottom-up using the system's parser.

An important feature is the integration of the graphics device into design entry; by using a mouse or tablet to point to locations on the current design picture, the user can use the current mouse coordinates in the insertion of various text macros into the current source line, e.g. a coordinate pair `34,23`, or a more complex macro might use two sets of coordinates to insert `box(10,20,10,3)`.

2.4.3 Conclusions

Interactive ILAP was a successful attempt at producing a programming environment for a procedural language, and indicated the potential of using a graphics device during the composition of programs (i.e. the use of positional information from the mouse). It did not, however, solve some of the insecurities of ILAP itself, and was not itself a firm basis for further development. The major deficiency in the implementation of Interactive ILAP was its parsing scheme (table-driven recursive descent) which imposed severe restrictions on the class of grammars which could be used and had efficiency problems. The other problem with the implementation was the rather ad-hoc relationship between the analysis records provided by the parser and the semantic actions which operated on them. Any change to the grammar required significant hand re-coding of large sections of code. The next chapter is concerned with providing a more secure framework for the development of a VLSI programming environment.

Chapter 3

The Wright Editor Generator

3.1 Introduction

1. Parsing and Attribute Evaluation
2. Pretty Printing
3. Picture Construction
4. Cursor Movement

Figure 3–1: Editor Structure

Figure 3–1 shows the structure of a language-based graphical editor. Editors built by the Wright system are specified using attribute grammars [39] [71]. An attribute grammar allows the specification of both the language's syntax and static-semantics (e.g. type-correctness). In the case of languages which have an immediate graphical interpretation (e.g. symbolic layout languages), the attribute grammar can also be used to define the construction process which will yield the corresponding picture. Not only can the grammar be used to describe the building of pictures, but it can also be used to restrict the domain of allowable pictures (e.g. preventing ground wires to be connected to power wires in a circuit). The editor uses a single representation of the design (an attributed

syntax tree) which the user can view and edit as either a language construct or as a picture. By allowing editing operations through both these interfaces the system combines the advantages of conventional programming techniques (hierarchy, parameterisation, iteration, conditional evaluation, type security etc.) with the benefits afforded by graphical entry (icons, menus, pointing devices, etc.) as well as providing constant pictorial feedback on the progress of the design. The use of an incremental evaluator restricts re-computation of attributes to only those affected by an editing change.

3.2 Attribute Grammars

I begin the discussion of attribute grammars (AGs) with a few definitions and assume the usual notational conventions [1] [91].

Definition 1 *A context free grammar for the language L is a quadruple, $G = (N, T, P, Z)$ with*

N *the set of non terminals*

T *the set of terminals with N and T disjoint*

P *a finite subset of $N \times V^*$, the set of productions, where $V = N \cup T$*

Z *a distinguished non terminal, the start symbol.*

Definition 2 *An attribute grammar is a quadruple, $AG = (G, A, R, B)$ with*

G *a context free grammar*

$A = \bigcup_{X \in T \cup N} A(X)$ *is a finite set of attributes*

$R = \bigcup_{p \in P} R(p)$ *is a finite set of attribution rules*

$B = \bigcup_{p \in P} B(p)$ *is a finite set of conditions*

For each occurrence of X in the structure tree corresponding to a sentence of $L(G)$, at most one rule is applicable for the computation of each attribute $a \in A(X)$.

Definition 3 *For each $p : X_0 \rightarrow X_1 \ldots X_n \in P$ the set of* defining occurrences *of attributes AF is $AF(p) = \{X_i.a \mid X_i.a \leftarrow f(\cdots) \in R(p)\}$. An attribute $X.a$ is called* derived *or* synthesised *if there exists a production $p : X \rightarrow \chi$ and $X.a$ is in $AF(p)$; it is called* inherited *if there exists a production $q : Y \rightarrow \mu X \nu$ and $X.a \in AF(q)$.*

Definition 4 *Let the set of synthesised attribute occurrences be $S(X)$ and the set of inherited attribute occurrences be $I(X)$.*
$S(X) \cap I(X) = \emptyset, \quad S(X) \cup I(X) = A(X)$.
An AG is said to be in normal form *if for every $p : X_0 \rightarrow X_1 \ldots X_n \in P$ the values of the attribute occurrences in $S(X_0)$ and $I(X_{1 \leq k \leq n})$ are defined as functions of attribute occurrences in $I(X_0)$ and $S(X_{1 \leq k \leq n})$.*

At this stage it will prove useful to introduce the notation used by the Wright system. This is the binary arithmetic example of Knuth [39]:

```
Grammar Binary is

  Code [ %include "att.src" ]

  Lexicals _zero, _one , _dot;

  Synthesised B(value), L(value, length), N(value), A(value);

  Inherited B(scale), L(scale);

  Productions

    A -> N <value$0 = value$1>;

    B -> _zero <value$0 = 0> |
         _one  <value$0 = twotothe(scale$0)>;

    L -> B <value$0 = value$1>
           <length$0 = 1>
           <scale$1 = scale$0>;
```

```
L -> L B <value$0 = value$1 + value$2>
         <length$0 = length$1 + 1>
         <scale$2 = scale$0>
         <scale$1 = scale$0 + 1>;

N -> L <value$0 = value$1>
       <scale$1 = 0>;

N -> L _dot L <value$0 = value$1 + value$3>
              <scale$3 = -length$3>
              <scale$1 = 0>;

  end of productions
end of grammar
```

In the following discussion the non terminals _zero, _one, _dot will be indicated by their literal occurrences '0','1', '.' when used in example strings. The above definition is used to give a precise meaning to the set of strings which are admitted by the grammar Binary. For example, the string 1101.01 can be parsed to form the tree shown in Figure 3–2.

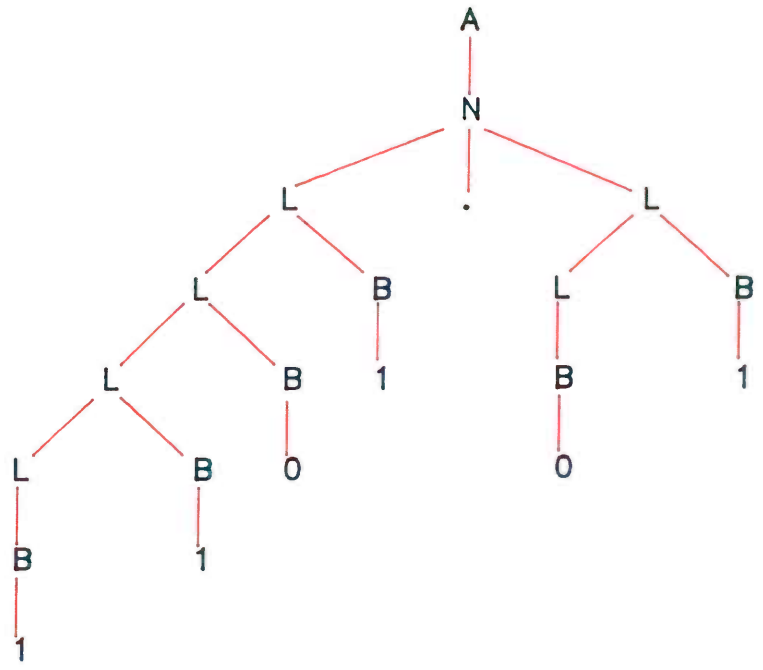


Figure 3–2: Syntax Tree for 1101.01

Meaning is assigned to this structure by evaluation of the attributes associated with the grammar's non terminals (e.g. non terminal L has *synthesised*

attributes `value` and `length`, non terminal `B` has *inherited* attribute `scale`). The brackets `< ... >` contain the *defining occurrences* of the AG, i.e. the semantic functions which link the attributes.

In the defining occurrence `<value$0 = value$1 + value$3>` for production `L -> L _dot B` , the $\$_n$ labels are used to indicate which subtree of the current production the attribute belongs to (note that this index also includes terminals in its calculation). All synthesised attributions can only be made at position 0 in the production, indicating movement of data *up* the syntax tree. In this example the 'values' synthesised for two subtrees are 'added' together. Also for this production there are the inherited attributions `<scale$3 = -length$0>` and `<scale$1 = 0>`. This is information flow *down* the syntax tree.

The defining occurrence `<value$0 = twotothe(scale$0)>` uses an auxiliary function `twotothe` which is just normal exponentiation 2^n. The actual form of the semantic actions is not covered by the formalism of attribute grammars, and the Wright system makes use of the IMP [73] programming language. The defining occurrences are just fragments of IMP code with attribute instances. The system replaces the attribute instances with appropriate references into the internal attributed syntax tree data structure. The above examples make use of IMP's arithmetic operators and assignment statement (indeed, the defining occurrences must always be of the form `<attr`$\$_n$ `= xyz >` where xyz is a valid IMP integer expression, optionally containing attribute instance references).

The example `Binary` has been chosen because of its brevity and because it illustrates the information flow possible in an AG. The grammar seeks to give an 'intuitive' meaning to binary strings which have a radix point (i.e. binary fractions). The above set of functions, when applied to the parse tree of our previous string `1101.01`, yields the attributed syntax tree shown in Figure 3–3.

The order in which defining occurrences are applied is determined by the evaluation scheme being used, and is discussed later. However, for the purposes of this example, observe that the `length` attributes to the right of the radix are required to be evaluated (bottom up) before the `scale` attributes can be

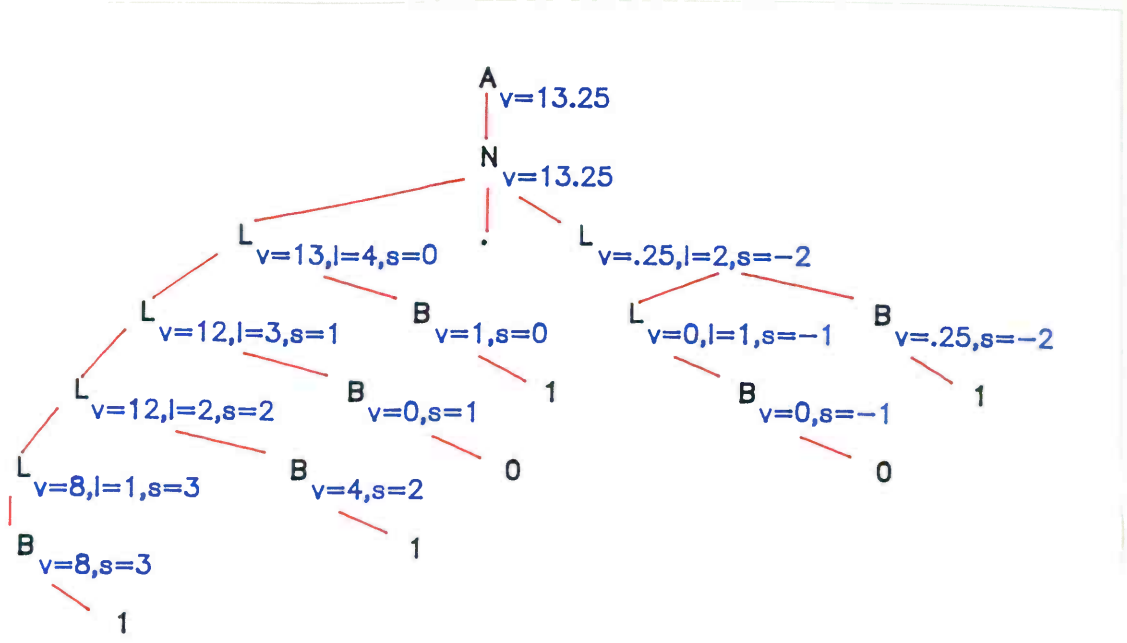


Figure 3–3: Attributed Syntax Tree for 1101.01

evaluated (top down). This allows the value attributes to be synthesised, and the 'meaning' of this tree determined, namely 13.25 (decimal).

This section has introduced the major features of attribute grammars and the Wright system input language. The following sections will describe how this specification technique can lead to the description of more complex (and useful) translations and the automatic generation of an interactive programming environment. The following chapters present larger examples of this formalism in action.

3.3 LALR Parsing

3.3.1 Introduction

The parser module used by the Wright system started life as the APG (Automatic Parser Generator) package [53]. This package was developed in the early

stages of the work described here, and was the first response to the experiences gained in implementing Interactive ILAP. Parsing technology is a crucial component in any translation process and so was the natural candidate for early investigation. The tools Lex [20] and Yacc [34] provided the kind of improvements needed on the Interactive ILAP scheme (this is discussed further after the tools have been described) but were not available on the computing resource I was using and do not conveniently fit in with the IMP programming environment. In order both to learn further the concepts involved and to provide a parsing facility for my computing environment, I designed and implemented an IMP parser generating system based on Lex and Yacc (ASG and APG) which I have been able to tailor exactly to my requirements.

LALR is the term given to the largest class of grammars which can be accepted by the APG system and is explained after a brief introduction to lexical analysis, as implemented by ASG.

3.3.2 Lexical Analysis

Lexical analysis is the process which builds *tokens* from the raw input provided in a program source. The split between syntax analyser and lexical analyser is fairly arbitrary (it would be possible to have the set of ascii characters as the tokens of the language, and use the parser to recognise numbers, identifiers, etc.) but it is usually convenient to let the lexical analyser build up groups of letters which are logically bound together (i.e. the components of a number), thus improving both the modularity and efficiency of the parsing process. In non-interactive systems the lexical analyser is also responsible for throwing away insignificant formatting characters and comments. Tokens correspond to the terminals in a grammar description (e.g. the tokens `_one`, `_zero` and `_dot` of AG `Binary`). For the rest of this section on parsing I will illustrate the APG programs by showing the implementation of the expression evaluator `Calc`. Here is the lexical definition of `Calc`:

```
lexical_definition Calc is
```

```
  ranges
    @N is '0'..'9';              {numbers}
    @L is 'a'..'z' + 'A' .. 'Z' {letters}
    @B is 0 .. 16_20;            {white space}
    @E is 0 .. 127 - '}'         {comments}
  end of ranges

  macros
    #case is $$;
    #p    is $($)*              {1 or more operator}
  end of macros

  expressions

    #case;

    _keyword -> \keyword;     {_keyword is case insensitive}

    #;

    _equals -> \=;
    _plus   -> \+;
    _minus  -> \-;
    _rb     -> \);
    _lb     -> \(;
    _div    -> \/;
    _times  -> \*;
    _form   -> #p[@B];
    _int    -> #p[@N]
               ( _ #p[ @N | @L ] | );
    _comm   -> \{@E*\}

  end of expressions

end of lexical_definition
```

The ASG specification consists of the declaration of *ranges*, *macros* and *expressions*:

Ranges Ranges are sequences of characters or combinations of ranges. Supported combinations are addition + and subtraction −. Ranges can be specified as lists of characters between double quotes, single characters between single quotes or ascii code ranges (e.g. `0..32`). Range names can be used anywhere in the definitions where a character could appear.

Macros The expression defining _form uses the macro #p. In the definition of #p the $ is a substitution operator which corresponds to the argument enclosed by [..]. Hence #p[@B] is expanded out into @B(@B)* in the definition body. The operator $$ substitutes a case conversion pair; #case[a] becomes (a|A), #case[A] becomes (A|a) and #case[,] becomes ,. Macros can also be used without parameters between expression definitions, causing every following character to have the macro applied to it. Global macro application of this nature is applied using a stack, which can be popped using the null macro #. The spurious definition keyword is inserted to show this working.

Expressions The tokens are defined using sequences of characters and ranges grouped by parenthesis. The following operators are available:

* post-fix operator which means "zero or more".

| infix operator which separates alternatives.

() denotes the null expression.

Lexical Conventions The character \ indicates that the following character is to be interpreted literally (thus allowing the definition of keywords and punctuation which are reserved ASG tokens). In all APG systems it is the convention to indicate that a name refers to a terminal by prefixing the character _ to the identifier (as in _int).

The underlying formalism in ASG is *regular expressions.* A definition from above (with macros expanded):

```
_int    -> @N @N* ( _ ( @N | @L ) ( @N | @L)* | );
```

can be read as *associate with the name* _int *the set of strings which consist of one number followed by zero or more numbers optionally followed by the radix character* _ *and a non zero sequence of numbers or letters.* The set of regular expressions

which together describe the complete token vocabulary are converted into a *deterministic finite automaton* which is implemented as a compacted transition table. Details of both the theory and this construction process are given by Aho and Ullman [1]. The lexical scanner operates as follows: each character from the input stream is used as an index into the transition table `T(character,state)` at the current state. The content of this position in the table is the next state. If this state is the error state then either a token has been recognised (i.e. the present state is an *accept state*) or there has been a lexical error, and a warning is issued.

Lexical scanners specified by regular expressions have several advantages over their hand crafted counterparts:

- it is very easy to add definitions or modify existing ones.
- the only code requiring maintenance is the generic system modules, hence an improvement or new feature to the scanner is immediately available to all system users.
- it is easier to define complex tokens *correctly.*
- the specification also provides a source for system documentation.
- the state transition table can form the basis of an auto-completion facility, i.e. once the user has typed a few characters of a token (in an interactive session), the system may be able to automatically complete the token, or provide a menu of possible completions.

Although not inherent in the formalism of regular expressions (which are often used as the specification for hand crafted scanners) but rather a feature of many automatic systems like Lex and ASG, is the use of large data structures like the transition table representation. Although the table is compacted (with a consequential run time speed penalty), equivalent hand crafted scanners can be expected to have less space requirements and to run faster. For interactive systems and prototype systems, space/time efficiency considerations are outweighed

by the previously described benefits. Even batch oriented systems are not significantly impaired by using an ASG scanner (applications of the APG/ASG system are presented later).

ASG is actually unmodified for use in the Wright system, the AG `Binary` has the trivial ASG description:

```
Lexical_Definition Binary Is

  Ranges
    @B Is 0..32;
    @C Is 0..127 - '}';
  End Of Ranges

  Expressions

    _zero     -> \0;
    _one      -> \1;
    _dot      -> \.;

    _form     -> @B@B*;      {white space}
    _comm     -> \{@C*\};    {comments like this one!}

  End Of Expressions

End Of Lexical_Definition
```

3.3.3 Syntax Analysis

I continue the development of the expression evaluator `Calc` by giving its APG definition:

```
Grammar Calc is

  Code [%include "calc.src"]

  Lexicals _int  [lex int],
           _plus, _times, _minus,  _div,
           _equals,
           _lb, _rb;

  Productions

    exp_list -> exp_list ans | ans ;

    ans -> exp _equals
           [print string("result = ")
            write(pop,0)
```

```
            newline]                                    |
           _error
           [print string("finger trouble!")
            newline];

    exp -> _minus exp      [push(-pop)]                 |
           _lb exp _rb                                  |
           exp _times exp  [push(pop*pop)]              |
           exp _plus  exp  [push(pop+pop)]              |
           exp _minus exp  [push(-pop+pop)]             |
           exp _div   exp  [push(div(pop,pop))]         |
           _int            [do int];

  End of Productions

  Priorities  (_times, _div) (_plus, _minus);

End of Grammar
```

An APG specification consists of the declaration of `Code`, `Lexicals`, `Productions`, `Priorities` and `Associativities` (not used in Calc).

Lexicals The names in the lexical list correspond to the regular expression definitions in the ASG specification.

Productions The grammar is specified using a variant of BNF, which includes the normal extension of parenthesis (..) followed by one of the operators ?, *, + for *optional* , *zero or more* and *one or more* groups of terminals and non terminals. This extension is not yet used by the Wright system, however, analogous notational extensions to AGs have been proposed [37], and would be a useful development.

Code Sections of the IMP programming language can be included within [..] brackets in three places; following the `Code` construct, at the end of each terminal definition and at the end of every production. The first piece of code is usually a set of declarations used later (often contained in a file referenced by the IMP `%include` statement). The code placed after lexical items is executed after the scanner passes the current token to the parser. The code at the end of production definitions is executed after the production is *reduced.*

Priorities and Associativities APG allows the use of *ambiguous grammars* by allowing the user to specify disambiguating relationships (e.g. multiplication and division have the same *precedence* but have higher precedence than addition and subtraction).

In order to explain some of the terminology introduced above, I now will briefly discuss the underlying mechanism of APG, namely LALR parsing.

In the same way that regular expressions can be implemented using finite automata, a subset of the context free grammars can be implemented using *deterministic pushdown automata* (DPDA). Informally, a DPDA can be regarded as a finite automaton (i.e. transition table) and an associated stack. A method of parsing which makes explicit use of this formalism is LR parsing [38], so called because the parser scans the input from left to right and constructs a rightmost derivation in reverse. Figure 3–4 shows the main components of an LR parser.

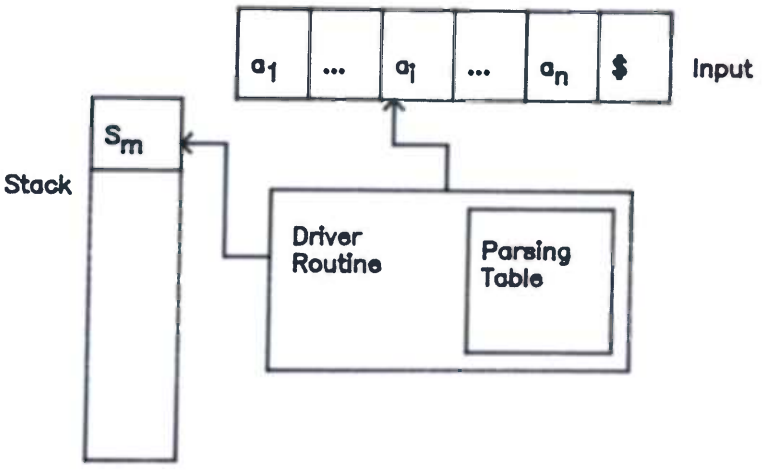


Figure 3–4: LR Parser

The parse tables consist of two parts `Action(token,state)` and `Goto(non terminal,state)`. The parser operates as follows: as each token is provided by the scanner an entry in the table `Action(token,state)` is looked up (initially at state zero). The contents of this location can be one of:

"shift *s*" push current input symbol onto the stack and the next state *s*.

"reduce $A \rightarrow \beta$" pop the symbols off the stack corresponding to this production and execute the code associated with this reduction. Push A onto the stack and push `Goto(`s`, `A`)`, where s is the new top of the stack.

"accept" Parsing completed.

"error" Syntax error, issue warning and perform error recovery.

For the grammar `Calc` and the input string `3 + 4 * 5` the parser will cause the sequence of stack operations shown in Figure 3–5.

	Stack	Pending Input
1	0	3 + 4 * 5
2	0 3 1	+ 4 * 5
3	0 Exp_7 5	+ 4 * 5
4	0 Exp_7 5 + 8	4 * 5
5	0 Exp_7 5 + 8 4 1	* 5
6	0 Exp_7 5 + 8 Exp_7 13	* 5
7	0 Exp_7 5 + 8 Exp_7 13 * 9	5
8	0 Exp_7 5 + 8 Exp_7 13 * 9 5 1	
9	0 Exp_7 5 + 8 Exp_7 13 * 9 Exp_7 14	
10	0 Exp_7 5 + 8 Exp_3 13	
11	0 Exp_4 5	
12	0 Ans 4	

Figure 3–5: Stack Operations for `3 + 4 * 5`

The actual parsing mechanism is very straight forward to implement, the major task in building an LR parser is determining the contents of the parse tables. This would be an extremely tedious and difficult task to do by hand for all but the most trivial grammars, and so automatic tools must be used. The method used by APG is LALR (*lookahead*-LR), again, the theory and some

implementation strategies for using this technique can be found in Aho and Ullman's book [1].

An essential stage in the construction of the parsing tables `Action` and `Goto` is building a data structure known as the sets of LR(0) items, illustrated in Figure 3.6 The item labels `I0,I1 .. I16` correspond to the state field in the parse tables. Items are marked with a . to indicate how much of their production has been processed, and the possible transitions from that state are shown at the end of the item. For example, `I7` has a LR(0) item which corresponds to the production `exp -> _lb exp _rb` reaching the point `_rb`. If a `_rb` is the next move from `I7` then `I12` will become the next state.

The class of grammars which can be directly implemented from the LR(0) data structure are known as SLR (simple LR). LALR parsers extend the range of admissible grammars by calculating *lookahead symbols* for each LR(0) item. These lookahead symbols reduce the number of ambiguities that can arise when deriving the parse table from the LR(0) construction. Ambiguities arise in the `Action` table when it can not be determined which stack operation to perform:

shift/reduce conflicts At the present state it is possible to either shift on the current input symbol and enter a new state, or to recognise a production on the top off the stack and reduce it.

reduce/reduce conflicts Two or more different productions can be reduced.

Sometimes ambiguities are deliberately introduced and then disambiguated using the `Priority` and `Associativity` statements. Compare the previous example of `3 + 4 * 5` with the operations on the string `3 * 4 + 5` shown in Figure 3–7.

The stack does not grow as far in this example because the parser generator has decided (from the `Priority` statement) to make a reduction by Exp_3 (Exp → Exp `_times` Exp) at state `I14` instead of the equally possible shift on `_plus` to state `I8`, thus giving multiplication precedence over addition. The `Associativities` command allows the user to specify right or left associativities, e.g. in $3 + 4 - 2$

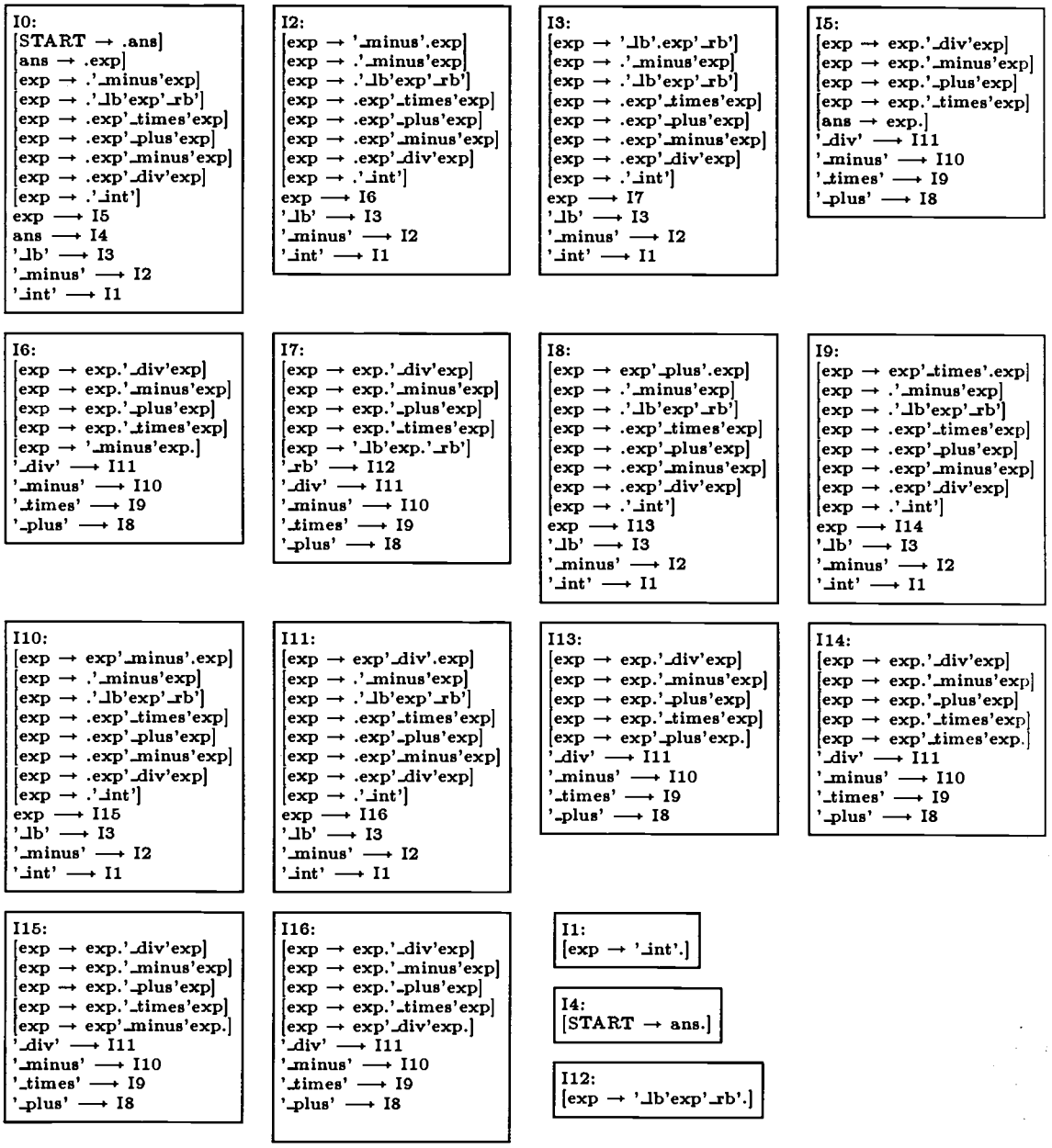


Figure 3–6: The sets of LR(0) items for grammar Calc

addition and subtraction have equal precedence, but since the default is left associativity, $3 + 4$ is recognised as an expression first, and not $4 - 2$.

Sometimes ambiguities arise because of poor grammar design or an incorrect grammar description. APG provides various diagnostic options which allow the parse tables (expanded), LR(0) sets of items, lookahead items, etc. to be printed

	Stack	Pending Input
1	0	3 * 4 + 5
2	0 3 1	* 4 + 5
3	0 Exp_7 5	* 4 + 5
4	0 Exp_7 5 * 9	4 + 5
5	0 Exp_7 5 * 9 4 1	+ 5
6	0 Exp_7 5 * 9 Exp_7 14	+ 5
7	0 Exp_3 5	+ 5
8	0 Exp_3 5 + 8	5
9	0 Exp_3 5 + 8 5 1	
10	0 Exp_3 5 + 8 Exp_7 13	
11	0 Exp_4 5	
12	0 Ans 4	

Figure 3–7: Stack Operations on 3 * 4 + 5

out. I have found this an invaluable aid in debugging both grammars and the system itself.

The reasons for choosing LR parsing can now be stated:

Wide Range of Grammars LR parsers cover the range of grammars which can be parsed using the major alternative method, recursive descent, and can admit many other grammars in addition. The ability to use both right and left recursion leads to succinct expression syntaxes, as does the ability to state operator precedences.

Errors LR parsers detect syntactic errors as soon as it is possible in a left-to-right scan of the input, and so are ideal for interactive entry. For batch-oriented applications many error recovery schemes have been developed which make use of the parse tables to attempt *intelligent* repair and recovery from errors. The APG system does not provide anything more sophisticated than a simple restart mechanism which is controlled by the

special terminal `_error` which the user puts as an alternative production on the syntactic entity on which parsing is to continue after a syntax error (i.e. tokens are skipped until $A \rightarrow$ `_error` is a valid reduction). This technique (known as *panic mode*) is similar to the *skip until semi-colon* error action of some Pascal compilers.

Table Driven The LALR(1) automaton is stored as a table; this simplifies the organisation of incremental parsing schemes by allowing un-expanded sections of the parse tree to record positions in the automaton which will later be used to restart parsing (this is described in more detail in the next chapter). Having the automaton readily available helps in the implementation of the error recovery schemes mentioned before, and in interactive systems makes the provision of error messages a trivial matter (i.e. whenever an error occurs it is possible to give a menu of legal alternatives, derived straight from the table). This technique could be extended on lines analagous to the auto-completion of lexical tokens, however, this is not currently part of the Wright generic parser.

Although not all used by APG, there are a great number of space/time optimisation techniques which can be employed to make LR parsers more efficient than they already are. APG parsers certainly perform well enough for the purposes of the Wright system, and indeed, many other systems.

At this moment APG has been in service for two years and has been ported to a number of machines and operating systems. Among the projects that have made use of it are:

- A microcode assembler.
- An IMP syntax analyser.
- A silicon compiler (Chip Churn [64]).
- An OCCAM to hardware compiler [52].

- APG (isn't bootstrapping wonderful!).
- A hardware description language based on Pascal.
- An EDIF [19] syntax analyser (which had 605 productions !)

While not attempting to fully explain LALR parsing or parser generation, this section has introduced the basic structure (the LALR(1) automaton) on which the Wright System bases its parsing functions, and the motivation for this choice. The APG module and its grammar description language has been incorporated into the Wright system with only one major difference; the addition of attribute declarations and defining occurrences. The major contribution of APG to the Wright System has been its provision of a framework for *bottom-up* program entry during sub-tree replacement.

3.4 Attribute Evaluation

The attribute grammar's defining occurrences are a *declarative* specification of the syntax tree's semantics; they do not imply any specific order of evaluation, other than the obvious restriction that a function can not be evaluated until all its arguments have been evaluated. There are many ways of organising the evaluation phase, the method used in Wright was chosen because of its simplicity (it can be implemented fairly quickly) and also because the described implementation supported incremental evaluation. Before giving a brief overview of the algorithm it is necessary to make another two definitions:

Definition 5 *For each* $p : X_0 \rightarrow X_1 \ldots X_n \in P$ *the set of* direct attribute dependencies *is given by:*

$$DDP(p) = \{(X_i.a, X_j.b) | X_j.b \leftarrow f(\ldots X_i.a \ldots) \in R(p)\}$$

Definition 6 *For an attributed syntax tree* S *with nodes* $K_0 \ldots K_n$ *corresponding to application of* $p : X_0 \rightarrow X_1 \ldots X_n$, *the* dependency tree relation *is given by the set:*

$$DT(S) = \{(K_i.a, K_j.b) | (X_i.a, X_j.b) \in DDP(p)\}$$

where we consider all applications of productions in S.

The direct attribute dependency sets (which can be directly derived from the attribute grammar specification) give the dependencies between attributes in a single production. The dependency tree relation gives the complete attribute dependencies for any given parse tree, and can be viewed as the gluing together of the DDPs of that tree. If this relation contains a cycle, then the attribute grammar is said to be *circular.*

The Wright system uses an evaluation method described by Jalili [32] [31]. The algorithm starts by taking the synthesised attributes at the root of the parse tree and pushes them onto a stack of attributes pending evaluation. Evaluation can occur if the attribute at the top of the stack has no dependencies in its DDP (i.e. for evaluation of $X_j.b$ there must not exist $(X_i.a, X_j.b) \in$ DDP, or all its dependencies have already been computed). If evaluation can not occur then the dependent set of attributes (all the $X_i.a$) which have not been computed are pushed onto the stack. If an attribute has already been marked as having been pushed, then there is a circularity in the DT, and the evaluation fails. If evaluation of the attribute at the top of the stack can occur, then its semantic action is executed, the attribute is marked as being evaluated and it is popped off the stack. The algorithm continues until a circularity is detected or the stack becomes empty.

Figure 3–8 shows the order in which the attributes are calculated for the `Binary` example `1101.01` (the symbol '*' indicates that the attribute was not required to be evaluated).

The process being performed here is a topological sort of the DT relation by depth-first search, interleaved with attribute evaluation and circularity checking. The method is dynamic, with the DT relation never actually being constructed. A procedural version of this algorithm, which makes the stack implicit in its procedure calls, gives a more succinct description of this tree traversal (modified from an example by Engelfriet [50]):

for a synthesised:

Proc a_eval (K:node)
  Begin If Not evaluated(a,K)
    Then Let $p : X_0 \rightarrow X_1 \ldots X_n$ be the production at K.
      Let $X.a := f(\ldots X_i.b_j \ldots) \in AF(p)$
      $\ldots$; b_j_eval(K_i) ; $\ldots$
      $K.a := f(\ldots K_i.b_j \ldots)$
      evaluated(a,K) := true
    Fi
  End.

for a inherited

Proc a_eval (K:node)
  Begin If Not evaluated(a,K)
    Then Let FK be the father of K with production
      $p : X_0 \rightarrow X_1 \ldots X_n$
      where X_s labels the son corresponding to K.
      Let $X_s.a := f(\ldots X_i.b_j \ldots) \in AF(p)$
      $\ldots$; b_j_eval(FK_i) ; $\ldots$
      $K.a := f(\ldots FK_i.b_j \ldots)$
      evaluated(a,K) := true
    Fi
  End.

The major feature of this algorithm is that the DT relation is not actually constructed, dependencies are determined dynamically from the DDPs and the parse tree. The stack based version extends the *marking* method used to detect AG circularity by adding a timestamp field to the status field associated with each attribute. The evaluator uses the timestamp field and status field in such a way as to avoid un-necessary re-evaluation of attributes during interactive

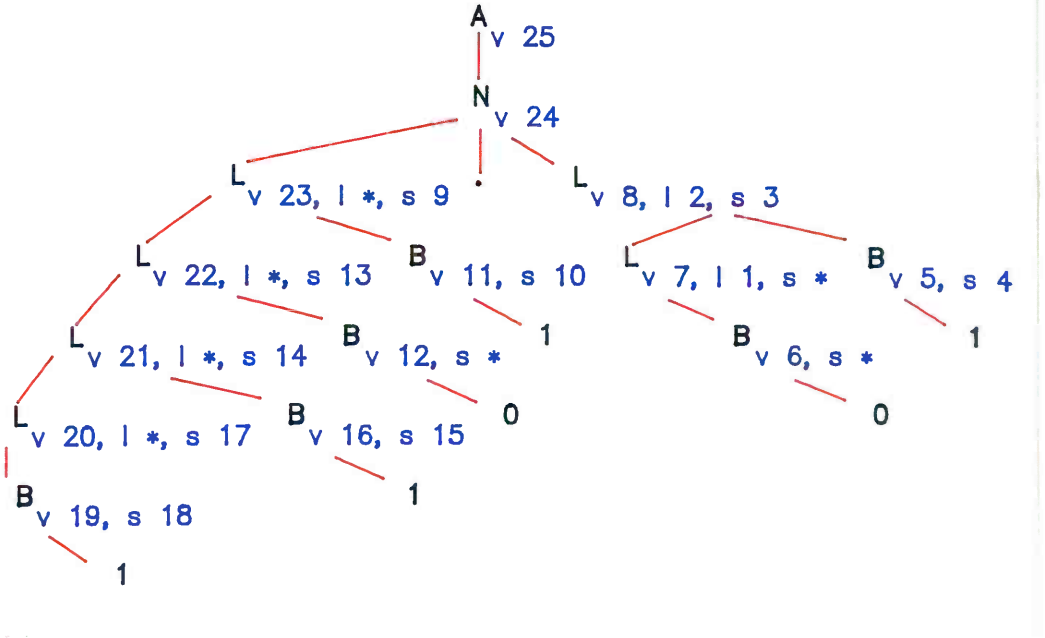


Figure 3–8: Order of Attribute Calculation for `1101.01`

tree editing, (i.e. it evaluates only those attributes affected by a given subtree replacement). The time complexity for the update is $O(N)$ where N is the number of attribute instances needed for static evaluation of the synthesised instances of the root of the semantic tree. Figure 3–9 shows the order and extent of attribute re-evaluation for the modification of the `Binary` example `1101.01` to `1110.01`.

The drawback to this algorithm is that although linear, it requires the traversal of the tree to start at the root, since this is the only way that changed attributes can be identified. In the Cornell Synthesizer Generator an incremental evaluation method has been developed which is linear in number of attributes which have actually been *affected* by the current sub-tree replacement. Since this algorithm is optimal in time, it would be a serious candidate for inclusion within the Wright system. However, the Jalili algorithm has enabled the Wright system to be produced quickly, and is sufficient for the purposes of demonstrating the AG specification technique.

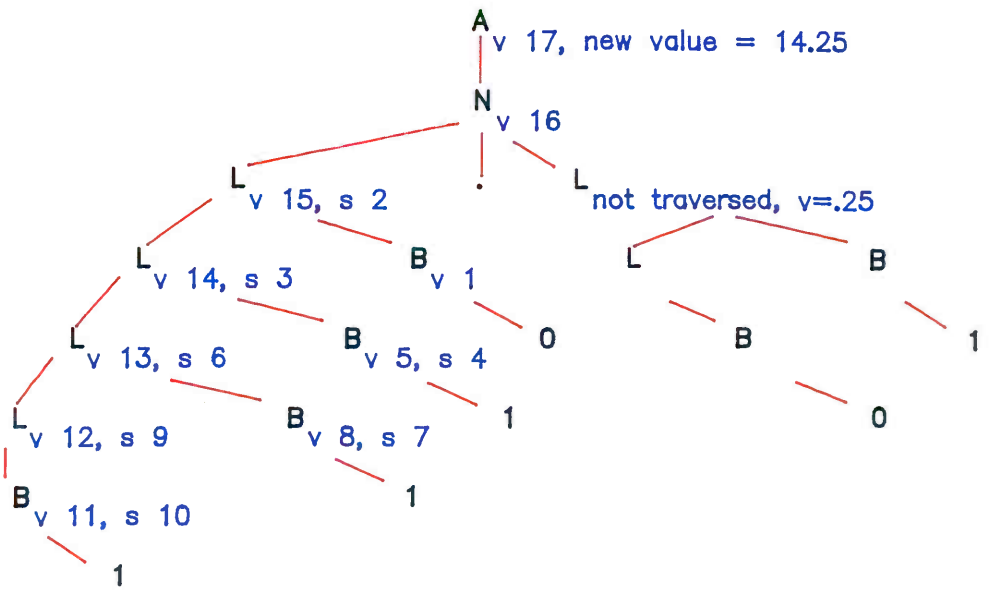


Figure 3–9: Changing a sub-tree

The availability of a superior algorithm highlights both a deficiency in the Wright system, and a strength of using high level specification techniques; systems making use of a formalism which is the area of active research have the potential to benefit from the discoveries made by that research. Any improvement made to the Wright evaluator module would benefit *every* editor specified for the system.

3.5 Structure Editing

3.5.1 Introduction

The previous sections have introduced the formalism of attribute grammars, shown the specification language for the system and described the underlying evaluation mechanism. This section will introduce the operational components of a Wright Editor; the window manager, the prettyprinter and the editing interface.

3.5.2 Window Management

The display functions of the Wright system are organised using a window manager specially developed for the project. The computing environment on which Wright was developed has excellent graphics hardware facilities, but very little software with which to exploit them. Wright's window manager (WM) was developed to provide support for the device configuration used in text/picture editing; namely, a video terminal and an 8-plane colour graphics display. Figure 3–10 shows this hardware configuration.

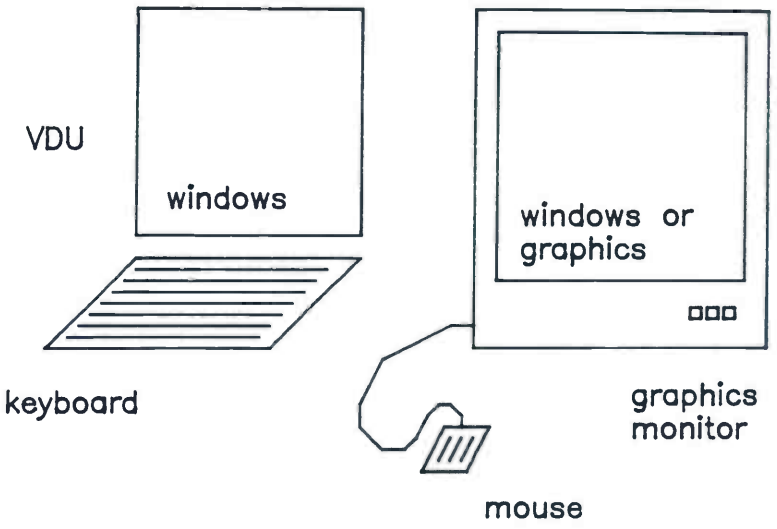


Figure 3–10: Device Configuration

Figure 3–11 shows the organisation of the WM's universe, known as *device space*. Windows and devices can be mapped arbitrarily onto device space and the devices will display any windows that intersect with them. Windows can overlap and are organised in a circular queue, the ones at the top overlaying those below. Windows can be moved about in device space and can also be moved about in the display queue. Devices can also be moved about in device space.

The windows can be accessed by application programs in two ways:

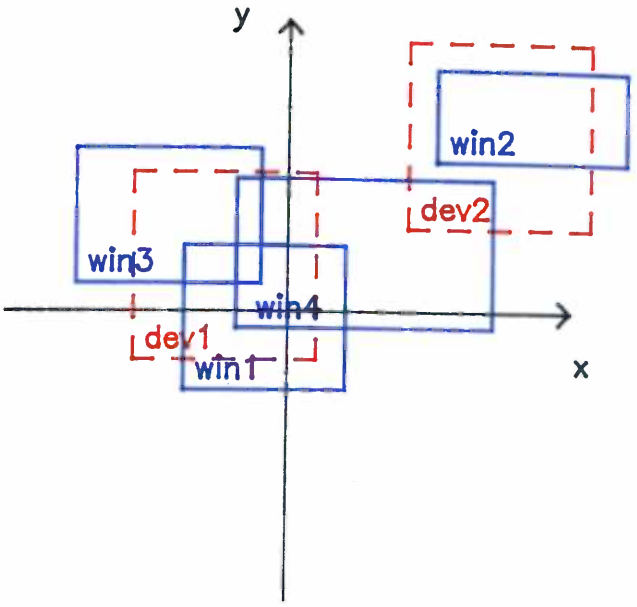


Figure 3–11: Devices and Windows

- normal system I/O routines are intercepted by the window manager which diverts the output to the currently selected window which behaves just like a video terminal would (scrolling, cursor movements, highlighting etc.)

- a memory mapped screen image (character based) which the applications program can manipulate directly.

The window manager also provides one special window which supports the graphics package EDWIN [29]. This window takes up one half of the graphics device's frame-store and has exclusive rights to it. Applications programs can cause pictures to appear in this window by calling Edwin graphics procedures. Other windows can appear in the other half of the frame-store, and a software switch allows either half to be instantly displayed using the whole of the display.

The current Wright Editor module makes use of four windows:

- a window for the syntax tree un-parsing (pretty-printing).

- a window for attribution error messages and system warnings.

- a window for command and text entry.

- the graphics window.

In addition, the WM has its own window "**window manager**" which can be invoked at any time to re-arrange device space or to escape to the operating system.

3.5.3 Pretty-Printing

Pretty-printing [65] [96] is the process of un-parsing the attributed syntax tree into its textual form on a display device. The aim of any un-parsing scheme is to present as much useful information as possible in the available space, which can range from a 80x24 screen to a full listing on paper. The control of what can appear in the available space is known as *holophrasting*. Although structure editing necessarily requires pretty-printing of some sort (since the user is manipulating tree structures, not lines of text), it is a useful topic in itself due to the increasing importance of program *readability* (whether using structure editors or not). Whereas the bulk of a program may be written only once, it is likely to be read and modified many times, possibly by more than one person (including the original author). Consistency in layout style can make this an easier task.

In the Wright system pretty-printing is performed by a tree-walking procedure which uses information yielded by the evaluation of a synthesised attribute which is associated with every non terminal in the parse tree. Each non terminal has a maximum width in which it can display its sub-components, overflows cause the non terminal to be displayed vertically with appropriate indentation. The grammar designer controls the operation of the pretty-printer by setting various parameters in each attribute occurrence of the pretty-printing attribute. This will be illustrated in the next chapter.

Specifying an un-parsing scheme using an attribute grammar follows naturally from the proposal by Rose and Welsh [74] that language definitions for programming languages should contain indications on how the language is to be formatted. This work is developed by Woodman [96] who presents a *formatted*

definition of Modula2, using Rose and Welsh's extension to BNF which includes formatting commands and has an associated pretty-printing algorithm. The approach taken in Wright is rather more restrictive, and suggestions for further developments are given in the final chapter.

3.5.4 Editing

The Wright editor provides the normal tree traversal commands found in structure editors and also provides an interactive parser. Appendix A. includes a summary of the available commands, and most of them will be introduced in the next two chapters.

In addition to tree editing operations, Wright supports graphical interaction. Graphical interaction can take two forms:

Graphics Editor Commands The Wright system allows the user to define editor commands and bind them to keys on the keypad. These commands may make use of information contained in the attributed syntax tree and also positional information provided by the graphics pointing device. In Stick-Wright a command of this nature is defined which moves the system cursor to a position in the abstract syntax tree determined by the position of the pointing device on the current display image.

Graphics Text Macro Insertion In a similar manner to the mechanism described above, the user can define text insertion macros which can be invoked during the parsing of a sub-tree replacement. This macro can also make use of the attributed syntax tree and positions from the pointing device. In Pict-Wright a text insertion macro is defined for inserting co-ordinates into the program script.

These activities are further explained in the example editing sessions presented in the next two chapters.

3.6 Summary

A Wright specified editor consists of the following components:

- a lexical analyser specified by regular expressions
- a syntax analyser specified by an LALR grammar
- a set of semantic functions specified by an attribute grammar
- an incremental attribute evaluation algorithm
- a window manager
- a graphics package
- a pretty-printer
- an interactive parser
- a tree editor
- a set of graphical commands

The following chapters illustrate these components in action.

Chapter 4

Pict-Wright

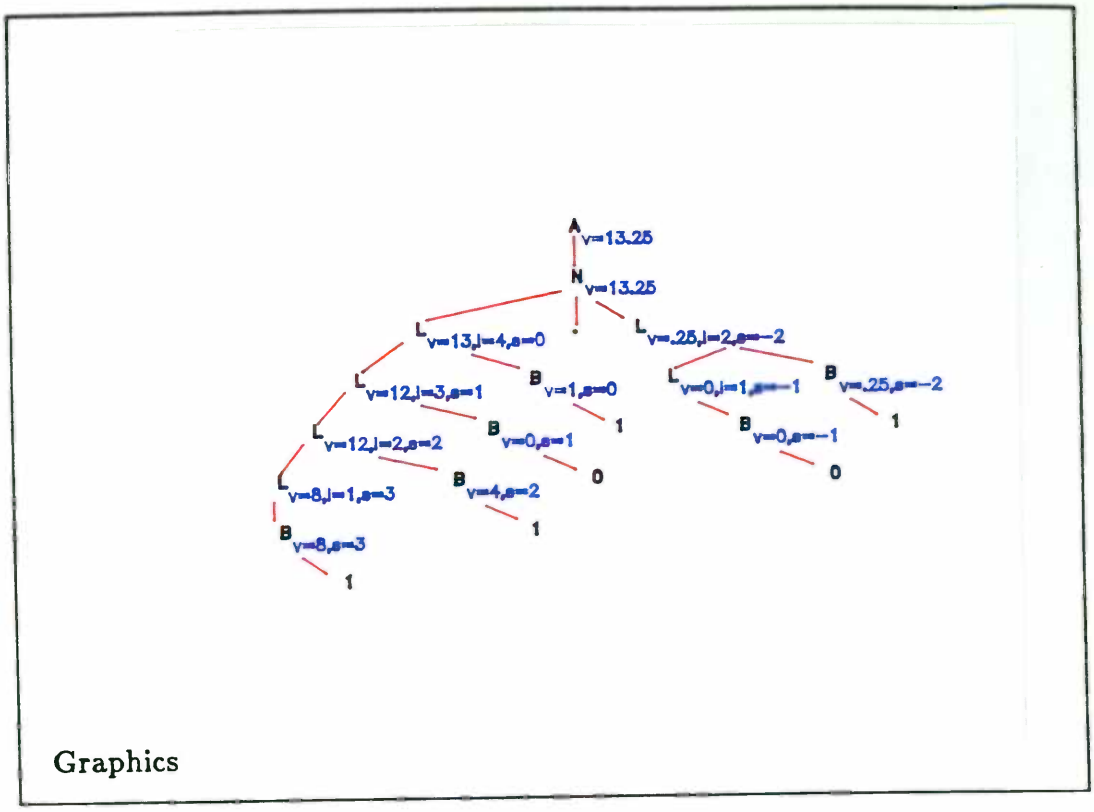
4.1 Introduction

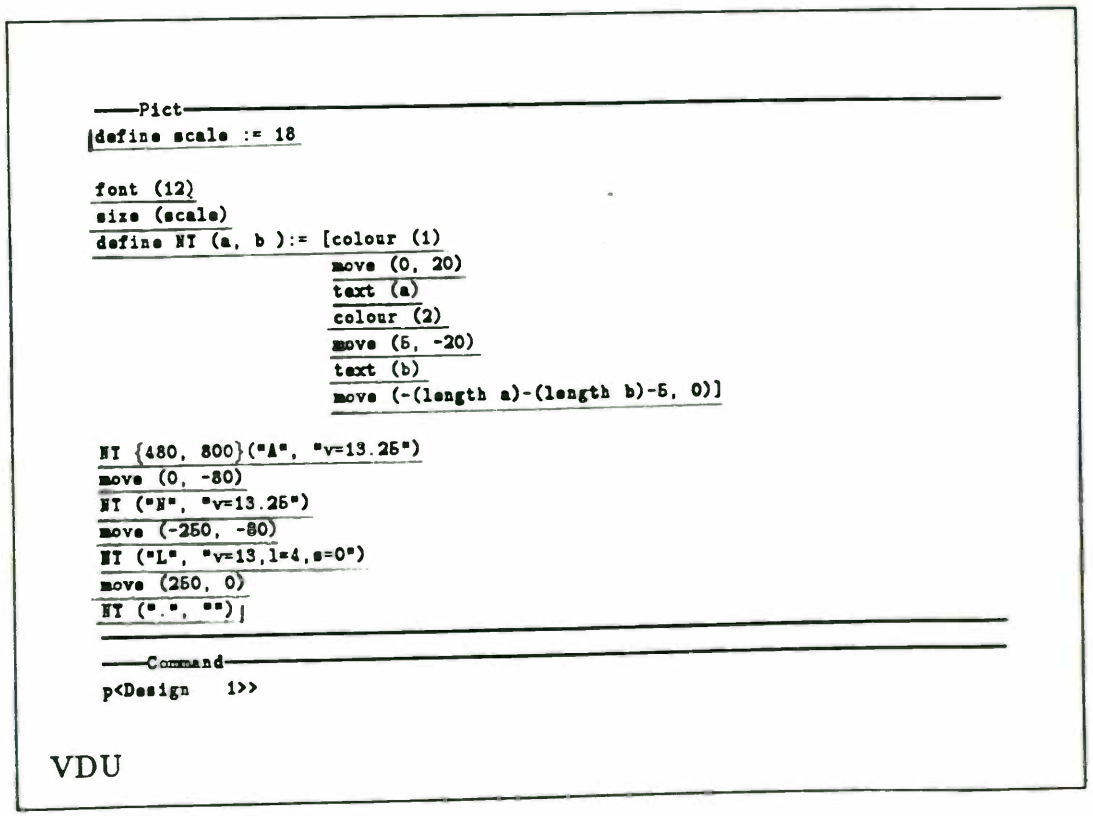
Pict-Wright is a simple picture editing system built to demonstrate the efficacy of attribute grammar specification techniques to interactive editing and picture generation. While the grammar is small, and hence more easily described, the techniques used are powerful and have a much wider range of applications. A no less important reason for building the system was that it enabled the generation of many of the figures that appear in this thesis. Figure 4–1 shows a snap-shot of a Pict-Wright editing session (using an illustration taken from the last chapter).

The textual interface of Pict-Wright consists of a simple imperative language which uses the line and text graphics primitives provided by the Edwin [29] graphics package. The language allows parameterised groups of graphics commands to be bundled into procedures. Within each command procedure (including the top level list of commands) there is the notion of a *current position.* This is defined to be the place where the last graphics command finished drawing, and is the default starting position for the following command. Each drawing command takes the following form:

```
action {x, y} (a, b)
```

where `action` is the name of the drawing command, `{x, y}` are the coordinates of the command's local origin (the default is the current position) and `(a, b)` are the command's parameters (there may be any number and type of parameters).





VDU

Figure 4–1: The Pict-Wright Editor

Parameter values can be integer expressions or string expressions (the grammar includes type-checking rules for preventing illegal expressions, e.g. 2 + "two".

In Figure 4–1 the picture being displayed (shown in greater detail on page 42) is constructed as a list of procedure calls separated by relative move commands. The first procedure call:

```
NT {480, 800}("A", "v=13.25")
```

fixes the root of the tree being depicted in this figure. The final screen position {480, 800} went through several iterations as I interactively discovered how the tree was growing. The connecting lines in the figure were all instantiated using a *graphical insertion macro.* The macro used to draw lines takes two coordinates (the end-points of the desired line) which are provided by the user via the graphics pointing device. The Pict-Wright statement that will cause the line to be drawn is then inserted into the current text-window.

Pict-Wright is not a full implementation of an imperative programming language, and has several constraining omissions (e.g. no condition or loop statements). Pict-Wright was implemented in an evolutionary manner (i.e. new primitives were added as I found them necessary), and the extension of the language to include more features and control structures should pose no serious problems. The features currently provided, however, seem to be sufficient for the production of simple line/text drawings.

This chapter proceeds by introducing the lexical and syntactic aspects of Pict-Wright, in preparation for the explanation of the attributes and attribution rules which govern Pict-Wright's operation. An example of a Pict-Wright editing session is then followed through.

4.2 Lexical Definition

The following text is the specification for Pict-Wright's lexical analyser (and is the input for the Wright scanner generator ASG):

```
Lexical_definition pict is

  Ranges
    @L is 'a' .. 'z' + 'A' .. 'Z';
    @N is '0' .. '9';
    @B is 0 .. 32;
    @S is 0 .. 127 - '"';
    @NotNL is 0 ..127 - 10;
  end of ranges

  macros
    #case is $$;
  end of macros

  expressions

          #case;
            _define -> \define;
            _line   -> \line;
            _colour -> \colour;
            _font   -> \font;
            _size   -> \size;
            _text   -> \text;
            _length -> \length;
            _move   -> \move;
          #;

          _lb     -> \(;
          _rb     -> \);
          _slb    -> \[;
          _srb    -> \];
          _clb    -> \{;
          _crb    -> \};
          _comma  -> \,;
          _ass    -> \:=;
          _minus  -> \-;
          _plus   -> \+;
          _times  -> \*;
          _div    -> \/;

          _id     -> @L(@L|@N)*;
          _int    -> @N@N* {was that a radix ?}
                     ( {yes} _(@N|@L) ! (@N|@L)* | {no} );
```

```
        _string -> \"@S*\"(\"@S*\")*;

        _blank -> @B@B*;

        _comment -> \\\-@NotNL*\
                    ;

  end of expressions

end of lexical_definition
```

The above definition introduces Pict-Wright's eight keywords (`define`, `line`, `colour`, `font` , `size` , `text`, `length` and `move`) all of which are case insensitive, the delimiters and separator (`( ) [ ] { } ,`), the assignment operator `:=`, the arithmetic operators (`+ - * /`) and identifiers, integers, strings, formatting characters and comments. From this specification a complete lexical analyser is generated.

Notice that the definition of `_string` allows strings of the form:

```
"this is a single-double-quote -> "" <- "
```

which is later interpreted by the system as being the string:

```
this is a single-double-quote -> " <-
```

The definition _int allows the optional specification of a number base, the default being decimal (e.g. 16_A0 is interpreted as the decimal integer 160).

The following statistics are printed by ASG during the construction of the lexical analyser:

```
ASG: parsing complete with no errors in 1277ms
ASG: DFA took 32844ms to build
ASG: DFA took 868ms to minimise
ASG:Compact: old size = 62*58 = 3596 entries
ASG:Compact: new size = 62*2 + 309*2 = 742 entries (38 misses)
             in 8924ms
```

This shows the times taken for various operations (the total construction time, including program loading and file writing, being under a minute). The

DFA (deterministic finite automaton) has a 62 character alphabet and 58 states. The space requirements for representing this are reduced by a table compaction procedure to 742 entries (a miss is a redundant entry in the compacted representation).

While the output from ASG could form the basis of an operational scanner, it still has unexpanded ranges (i.e. alphabet characters which represent ranges of characters) and has no knowledge of the tokens expected by the syntactic stage. A *linking* stage between the scanner and parser, implemented by the program HARD, changes the alphabet to include all the ascii characters 1 ...127, and expands the range characters of the ASG DFA into this new character set (e.g. transitions previously entered for character range `@N` are now entered for all the characters '0' ... '9'). The final task of the HARD program is to work out which token from the parser the scanner has recognised: it does this by comparing the names of the definitions in the ASG specification and the `Lexical` names in the Wright grammar specification. The following statistics are issued by HARD:

```
HARD: scanner token _blank will be ignored by parser
HARD: scanner token _comment will be ignored by parser
HARD:Compact: old size = 62*127 = 7874 entries
HARD:Compact: new size = 62*2 + 990*2 = 2104 entries (44 misses)
              in 38522ms
```

The scanner tokens `_blank` and `_comment` are not in the grammar, and are discarded by the scanner. This has the implication that comments can not survive editing sessions, unless they are explicitly made part of the grammar specification. The Wright system therefore makes comments a direct responsibility of the editor designer, who must determine where they can occur in a program text by specifying legal positions for them in the grammar. There are other approaches to dealing with comments so that they can appear anywhere a token can begin; in Interactive ILAP [54] comments were extracted from the current line (during lexical analysis) and then positioned with right justification at the edge of the screen. Another feature of this system was that some comments were automatically introduced, e.g. modification times in the program header and procedure names repeated at the end of their definition. In

Wright editors these kinds of activities are not *built in* and must be specified by the editor designer through the attribute grammar and the user defined editing commands.

4.3 Syntactic Definition

This section contains the syntax definition part of the AG, the attribution rules will be introduced later. The complete grammar specification is given in Appendix B.

```
Grammar Pict is

  Productions

    Design -> CommandList;

    CommandList -> CommandList Command |
                   Command;

    Command -> _define _id ArgL Defn |
               _id Arg1 Arg2         |
               _line Arg1 Arg2       |
               _colour Arg2          |
               _size Arg2            |
               _font Arg2            |
               _move Arg1 Arg2       |
               _text Arg1 Arg2;
```

From the above it can be seen that a Pict program is sequence of one or more commands. The first command associates an identifier `_id` with a definition contained in `Defn`. The second command is a invocation of a defined command, and the following commands are the drawing primitives.

```
    ArgL -> _lb NL _rb |;

    NL -> NL _comma _id | _id;

    Arg1 -> _clb List _crb |;

    Arg2 -> _lb List _rb |;

    List -> List _comma Item | Item;
```

These definitions introduce the argument syntax, an ArgL list can only contain identifiers, Arg1 and Arg2 have different brackets and can contain any kind of **Item**. All three types of argument list are optional.

```
Defn -> _ass _slb CommandList _srb |
        _ass Item;
```

A definition can associate an identifier with either a **CommandList** or an **Item**. The former is the procedure mechanism implemented in Pict-Wright.

```
Item -> _lb Item _rb       |
        _minus Item        |
        Item _times Item   |
        Item _div Item     |
        Item _plus Item    |
        Item _minus Item   |
        _length Item       |
        _id                |
        _int               |
        _string;
```

Pict-Wright has a single grammar construction for its expression syntax, namely the non terminal Item. Two data types are supported in the grammar; strings and integers. The operator _length is only meaningful when used on strings, the arithmetic operators (_times, _div, _minus) are only meaningful when used on integers, with _plus causing string concatenation when used on strings. I will later show how the attribute grammar ensures that only meaningful combinations are evaluated.

```
End of Productions

Priorities (_times, _div) (_plus, _minus);

End of Grammar
```

Finally the precedence rules for the Item operators are given: _times and _div have equal precedence and both have greater precedence than _plus and _minus.

This syntax part of the AG description is handled by the Wright parser generator module, which is derived from the APG program described in Chapter 3. The table compaction statistics issued by Wright are:

```
WRIGHT old size = 65*24 = 1560 entries
WRIGHT new size = 65*2 + 534*2 = 1198 entries (137 misses) in 4040ms

WRIGHT old size = 65*10 = 650 entries
WRIGHT new size = 65*2 + 50*2 = 230 entries (7 misses) in 1271ms
```

The first table is the Action table, the second is the Goto parse table. The total processing time for the whole grammar is about 1.5 minutes. Before explaining the semantics of the language, a short example will illustrate all the syntactic features defined above:

```
define scale := 18

font (0)
size (scale)
wrong (1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
       11, 12, 13, 114, 1555, 234234,
       345, 123123, 123132            )
define box (a, b, c ):= [colour (c)
                         line (a, 0)
                         line (0, b)
                         line (-a, 0)
                         line (0, -b)]

define textbox (c):= [box ((length c)+2*scale, 3*scale, 3)
                      colour (4)
                      text scale, scale(c)                ]

box (1000, 500, 2)
textbox {500, 100}("Pict-Wright")
move (100, 0)
textbox ("Stic-Wright")
```

4.4 Semantic Definition

4.4.1 The Attributes

This section outlines Pict-Wright's attributes by introducing the semantic domains. The semantic domains are not included in the formalism itself, but are defined as IMP record structures in the auxiliary definition file, which also includes the semantic functions used in the attribute occurrences (the statement

`%include "pict.src"` in the AG informs Wright where to find the auxiliary definitions).

A summary of the file `"pict.src"` is given with the complete AG in Appendix B. The attributes in the AG are introduced in the following non terminal declaration lists:

```
Synthesised
  Design  (box, def, pos),
  CommandList (box, def, pos),
  Command (box, def, pos),
  Arg1 (box,vals),
  Arg2 (box,vals),
  ArgL (box,def),
  NL   (box,def),
  List (box,vals),
  Defn (box,val),
  Item (box,val);
```

There are five synthesised attributes:

box This is the attribute used by the pretty-printer, and is a record structure containing information for the current non terminal's un-parsing.

```
%record %format Text Box Fm (%short x, y, last x,
                             %byte folds, extra, auto, indent)
```

`x` contains the x-dimension of the text box.

`y` contains the y-dimension of the text box.

`last x` contains the x-coordinate of the last entry in the text box.

`folds` marks grammar symbols for *folding*, i.e. growth in Y-direction.

`extra` marks grammar symbols needing extra space (e.g. space after keywords and identifiers (`define␣scale␣:= ...`).

`auto` marks grammar symbols requiring a forced line break (folding) after being printed.

`indent` marks grammar symbols requiring indentation after being folded.

def This attribute is a symbol table (environment) associating identifiers with definitions.

```
%record %format env fm (%string(*)%name id,
                        %integer val,
                        %record(env fm)%name split, next)
```

`id` contains a pointer to the identifier.

`val` contains the value associated with the identifier.

`split, next` are links in the symbol table.

`pos` This attribute is the current cursor position, relative to the current origin:

```
%record %format pos fm (%integer x, y)
```

`vals, val` These attributes contain the list of parameter (`name, value`) pairs provided in a procedure call.

Here are the inherited attribute declarations:

```
Inherited
  CommandList (env,origin),
  Command (env,origin),
  Arg1 (env),
  Arg2 (env),
  NL   (env),
  List (env),
  Defn (env,origin),
  Item (env);
```

There are two inherited attributes:

`env` This attribute contains the current *environment.* The environment is constructed from bindings between identifiers and definitions generated by the `def` attributions, and uses the same record structure.

`origin` This attribute contains the coordinates of the origin of the current `CommandList` (and uses the `pos` attribute's record structure).

In the parse tree maintained by the editor, non terminal nodes have a single storage location allocated for each of their attribute values. This storage location is made available to the semantic functions as an IMP `%integer` variable. For

simple attributes, such as integer expression values, the storage location can directly contain the attribute value. For more complex attributes the storage location is used to contain a pointer to a record structure built on the IMP heap. Whereas all the semantic functions in the file `"pict.src"` appear as `%integer %function ...`, the majority are returning the address of some record structure on the heap.

The decision to restrict the attribute occurrences to IMP `%integer` assignments and making all attributes appear as `%integer` values to Wright considerably simplified the implementation of those parts of the Wright system which deal with the semantic functions, however, it places the burden of type security on the user (i.e. the system cannot check whether the address supplied by a semantic function is a pointer to the appropriate record structure). Methods for improving the type security of the system are discussed in the final chapter.

4.4.2 Semantic Functions

The purpose of the attributes introduced in the last section will be made clearer by the explanation of a selection of the attribute occurrences contained in the AG:

```
Design -> CommandList
            <box$0 = c(box$1 ,default, 0,0,0)>          (1)
            <pos$0 = pos$1 >                            (2)
            <def$0 = def$1 >                            (3)
            <env$1 = initial environment >              (4)
            <origin$1 = new origin >                    (5)

Command -> _define _id ArgL Defn
              <box$0 = c2(box$3 ,box$4 ,80, 2_1100,
                                            2_0001,
                                            2_0011)>    (6)
              <env$4 = envadd(def$3 , env$0 )>          (7)
              <origin$4 = origin$0 >                    (8)
              <pos$0 = origin$0 >                       (9)
              <def$0 = do binding( val$4 )>            (10)
```

Here we have two sets of attribution occurrences for two productions of the grammar `Pict`. Nos. `(2,3,8,9)` are simple transfer rules which some AG no-

tations automatically assume if no other attribution is provided, however, in Wright they must be explicitly included. Nos. (4,5) provide the initial (empty) symbol table for the top-level `CommandList` and an initial `origin`, (0,0).

The semantic function `c` in (1) and the semantic function `c2` in (6) evaluate the pretty-printing attribute `box`:

```
%integer %function %spec c  (%integer box1,
                                      size, extra,
                                      auto, indent)

%integer %function %spec c2 (%integer box1, box2,
                                      size, extra,
                                      auto, indent)
```

Both functions return the address of a new record structure which will be the new value of `box$0`. This new structure is determined by the text boxes synthesised *lower* down in the syntax tree (`box1` for `c` and `box1, box2` for `c2`) and also by the parameters specified by the grammar designer:

`size` The maximum width of the text box for the current non terminal. For Pict-Wright, the default width (used in (1)) is 30.

`extra` Marks the grammar symbols requiring extra spacing (2_1100 indicates the first two grammar symbols out of four)

`auto` Marks the grammar symbols requiring auto line-breaking.

`indent` Marks the grammar symbols requiring indentation after being folded.

From the parameters given for the call of `c2` in (6) we can see that a definition `Command` requires space after its first two lexical symbols and indentation if line-breaking occurs in its subsequent non terminals. The auto-line break for the final grammar symbol ensures a blank line after every definition. The pretty-printing attributes do not themselves cause text to be printed, rather, they decorate the tree with information that is used by a tree-walking procedure which re-evaluates the contents of the Wright text-window after every cursor move. The

redrawing of the textual un-parsing makes use of the minimal-redraw algorithm implemented in the window manager (i.e. only changes in the current image have to be repaired). The example program script given on page 73 is an example of output from the pretty-printer using the style parameters given in (1) and (6).

Attribution No. (7) causes the symbol table for the Defn non terminal to be the addition of the current global symbol table env$0 and also the symbol table synthesised for its parameters def$0.

Attribution No. (10) creates a new symbol table entry which is the binding of the identifier _id to the value val$4 synthesised in the non-terminal Defn. This leads us to the attribution occurrences:

```
CommandList -> CommandList Command
                  <def$0 = defadd(def$1 , def$2 )>      (11)
                  <env$1 = env$0 >                      (12)
                  <env$2 = envadd(def$1 , env$0 )>      (13)
```

Bindings created by the definition Command are added to the inherited symbol table of subsequent Command non terminals (13). This flow of symbol bindings is illustrated in Figure 4–2.

The semantic function:

```
%integer %function %spec envadd (%integer env1, env2)
```

generates a pointer to a new symbol table link element which itself points to the either further link elements or a binding generated by a def attribution. The symbol table in Pict Wright is therefore implemented as a tree of linked identifier/value bindings.

The semantic function:

```
%integer %function %spec defadd (%integer def1, def2)
```

takes the symbol table def element contained in def2 and causes its link element to point at def1, thus creating a chain of identifier/value bindings (which

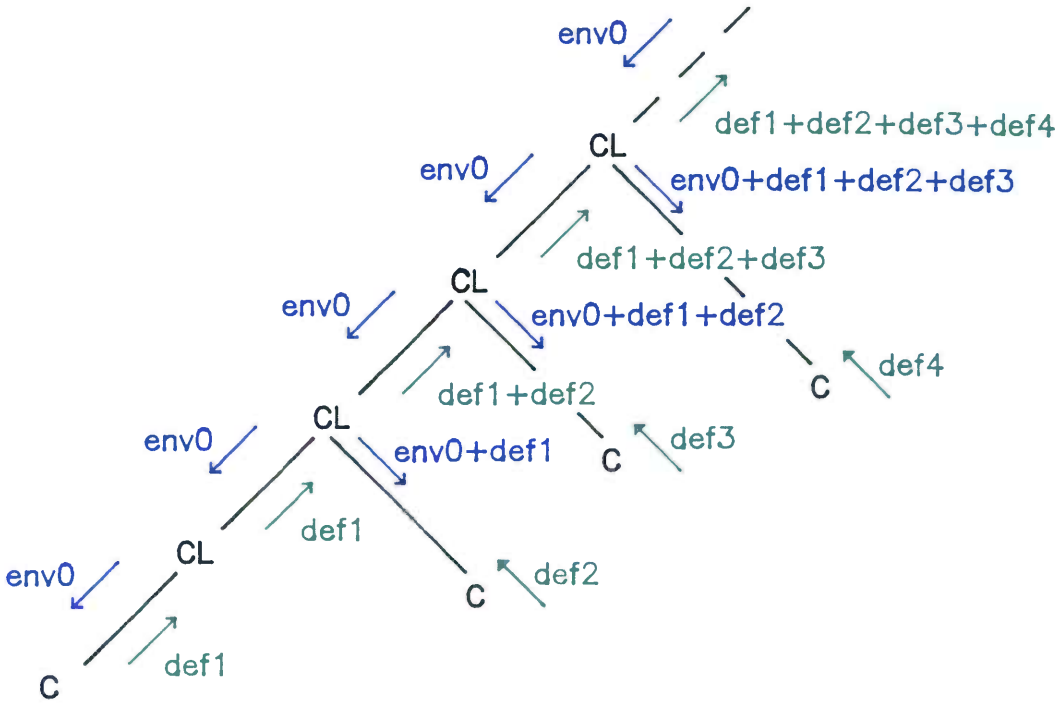


Figure 4–2: Symbol Table Attribute Flow

can then be inherited as env attributes by subsequent Command non terminals, as described above).

The symbol table structure as constructed above is made use of in the following Command and Defn attribution occurrences:

```
Command ->  _id Arg1 Arg2
              <env$2 = env$0 >                                  (14)
              <env$3 = env$0 >                                  (15)
              <pos$0 = do call(env$0, vals$2 , vals$3 ,
                               origin$0 )>                      (16)
              <def$0 = 0 {vals$2 vals$3 }>                      (17)
```

```
Defn -> _ass _slb CommandList _srb
          <val$0 = command ref(def$3 )>                    (18)
          <origin$3 = copy origin(origin$0 )>              (19)
          <env$3 = env$0 > |                               (20)

        _ass Item <env$2 = env$0 >                         (21)
                  <val$0 = val$2 >                         (22)
```

No. (16) is a drawing attribution which updates the current cursor position pos$0. The drawing operation is a recursive call of the attribution evaluator on the CommandList tree, which the binding of _id should reference (18). If _id is not in the current symbol table env$0, or is an Item value (22), an attribution error is reported[1].

The recursive evaluation (see also Figure 4–3) in (16) operates as follows: the arguments in Arg1 (or the current position if Arg1 is empty) are made available as an initial origin for the CommandList tree bound to _id (19), i.e. when copy origin is next called the coordinates provided in Arg1 will be given as the value of origin$3. The values given in the parameter list Arg2 are substituted into the environment synthesised for the arguments of the definition of _id (19) (def$3 in (7)). All the attributes in the CommandList tree are marked as being unevaluated. With the new origin and symbol table the synthesised attribute pos at the root of the CommandList is pushed onto the evaluation stack and the evaluator is then called.

This recursive evaluation technique for implementing procedure calling is a novel extension to normal AG practice and it led to the very fast development of the Pict-Wright editor. This is because there is no need to generate and store picture drawing code, pictures are drawn as a side effect of attribute evaluation. However, the method has a number of problems:

---

[1]The Wright-System does not directly implement attribute *conditions* supported by some AG systems, but the same effect is achieved by having condition checking and error reporting as a side effect of semantic functions

- it precludes the use of incremental evaluation (the *called* tree must have all its attributes marked as being unevaluated) and so looses the efficiency benefits of incremental compilation.

- it precludes the use of procedure recursion (unless multiple copies of the procedure's attributed syntax tree are to be made).

For a small system like Pict-Wright the efficiency problem is not a great concern (the system still operates fast enough to be interactive) and the lack of recursion has not prevented the description of fairly complex pictures.

The commented attributes `val$2` and `val$2` in `(17)` are there to cause evaluation of these argument lists during a `CommandList` procedure declaration (thus ensuring the checking of their identifier references, which would otherwise be delayed until the procedure was invoked).

So far no primitive drawing operations have been described, this is now remedied:

```
Command -> _text Arg1 Arg2
             <pos$0 = do text( vals$2, vals$3 , origin$0 )>    (23)
```

Attribution No. `(23)` cause the display of a text string contained in the parameter `vals$3` and at a position determined from `origin$0` and `vals$2`. The synthesised attribute `pos$0` is set to a coordinate at the end of the text string, relative to the current origin.

Pictures are generated in Pict-Wright as a side effect of the calculation of the synthesised attribute `pos`. Since an incremental evaluation scheme is being used (except during procedure call evaluation), it follows that subsequent editing to an initial attributed syntax tree would cause only the partial regeneration of the image (i.e. the portion just changed). This could cause anomalies since the Pict-Wright system does not know which areas of the screen image to erase before evaluating the new image. A brute force approach is taken to solve this problem; the origin for the whole picture is set as having been changed, causing

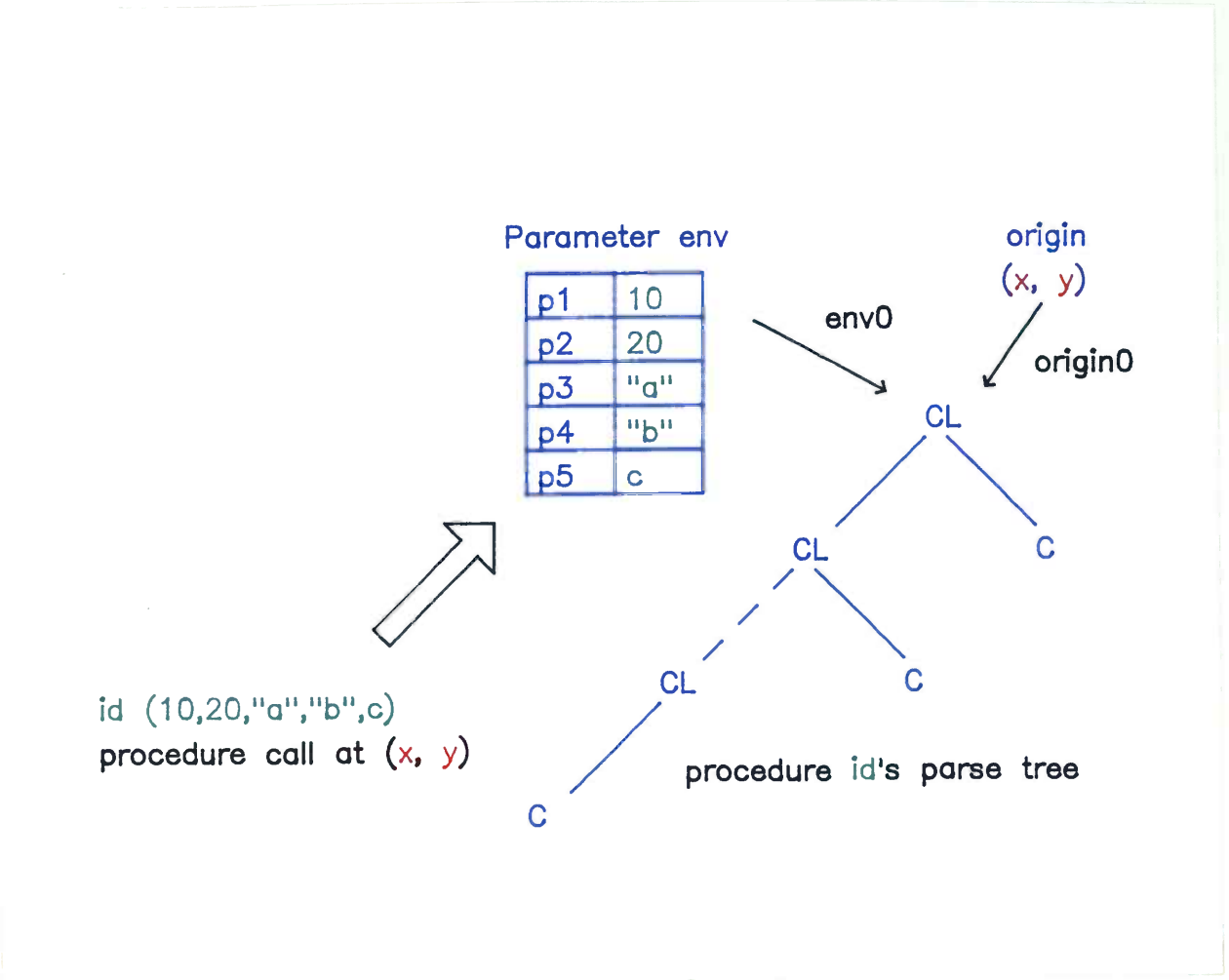


Figure 4–3: Procedure Call By Recursive Evaluation

the whole picture to be regenerated. In the chapter on Stick-Wright, a more efficient picture generation technique is presented which retains the incremental evaluation property lost by the recursive evaluation scheme used in Pict-Wright, and which also avoids unnecessary re-drawing.

The preceding *environment* attributions have used pointers to share common sections of symbol table, hence, each node in the syntax tree does not require its own copy of the symbol table. The linked list of identifier/value bindings is a rather inefficient representation since many identifier lookups would require long searches through the list structure until the desired binding was found. More

efficient representations for such structures do exist, e.g. Reps uses *shareable 2-3 trees* to implement symbol tables [71].

One problem with this *distributed* representation of symbol tables is that the distance between identifier declaration and usage can require a large number of copy rules to pass the attributes up and down in the parse tree. One proposed solution to describing remote relationships in attributed syntax trees, which avoids copy rules, is given by Johnson and Fischer [33], who present an extension to the AG formalism which permits non-syntactic attribute flow. This extension to the formalism allows distant nodes in the parse tree, which are closely connected semantically, to communicate directly. By leaving the standard AG formalism, however, they loose the ability to exploit the optimal incremental evaluation techniques as used in the Synthesizer Generator, and the automatic generation of editing systems is made more difficult by the introduction of the ad-hoc semantic relationships.

Finally, I discuss an expression attribution:

```
Item -> Item _plus Item
        <val$0 = do plus( val$1 , val$3 )>
```

The semantic function `do plus` checks whether `val$1` and `val$3` are both integers or both strings. Type can be determined from the value alone in Pict-Wright, since integers are constrained to be in a range less than the address value of strings, i.e. if a `val$1` is a valid IMP address, then the value of `val$1` is taken to be the string at that address, otherwise the value of the integer itself is taken. More complex data typing can be achieved by having a separate type field. Examples of type coercion and over-loading, and also further symbol-table schemes are given by Watt [50], a complete AG for ADA is presented by Uhl et al. [89].

In the first chapter the notion of design *malleability* was introduced. This property refers to a notation's suitability for manipulation using a syntax directed editor. Wright departs from conventional structure-editor wisdom by

basing the editing activities on the concrete syntax of the AG. Systems like the Syntheziser generator allow several concrete syntax specifications corresponding to an internal abstract parse-tree data-structure, on which the attributes are evaluated and stored. Wright uses the AG's concrete syntax without change mainly as an implementation expediency (it simplifies both the input specification and the integration of the parser with the editor). This has not proven to be a problem for the grammars developed for this thesis, but it did have implications on their design for malleability. This means in practice that the grammars have to be designed with their *screen appearance* firmly in mind. As the user traverses up and down the parse tree structure the current position is indicated by highlighting in the text window (`Pict`) and the current non terminal name is given in the `Command` window. The grammar designer must make sure that the structures being traversed are easy to follow, and form reasonable partitions. This is achieved by design choices like sticking to left-recursion for list structures (the user soon learns how lists of statements, ports, etc. are arranged) and using ambiguous expression grammars (with precedence rules), which form smaller trees than expression grammars using `Term` and `Factor` non terminals. Extra non terminals can be inserted to improve partioning between logically separated terms, which are otherwise left as siblings.

4.5 The Editor in Operation

I continue the development of the example presented in the syntax section; Figure 4–4 shows a picture of the system after processing the program script given on page 73.

The example contains a deliberate error, the procedure call `wrong`, indeed we can see that the system has detected this and has reported:

```
procedure wrong not declared
```

The `Command` window also shows which grammar construct the cursor is currently positioned on, `Design 1`, and the number of attribution errors, `>1>` , if any.

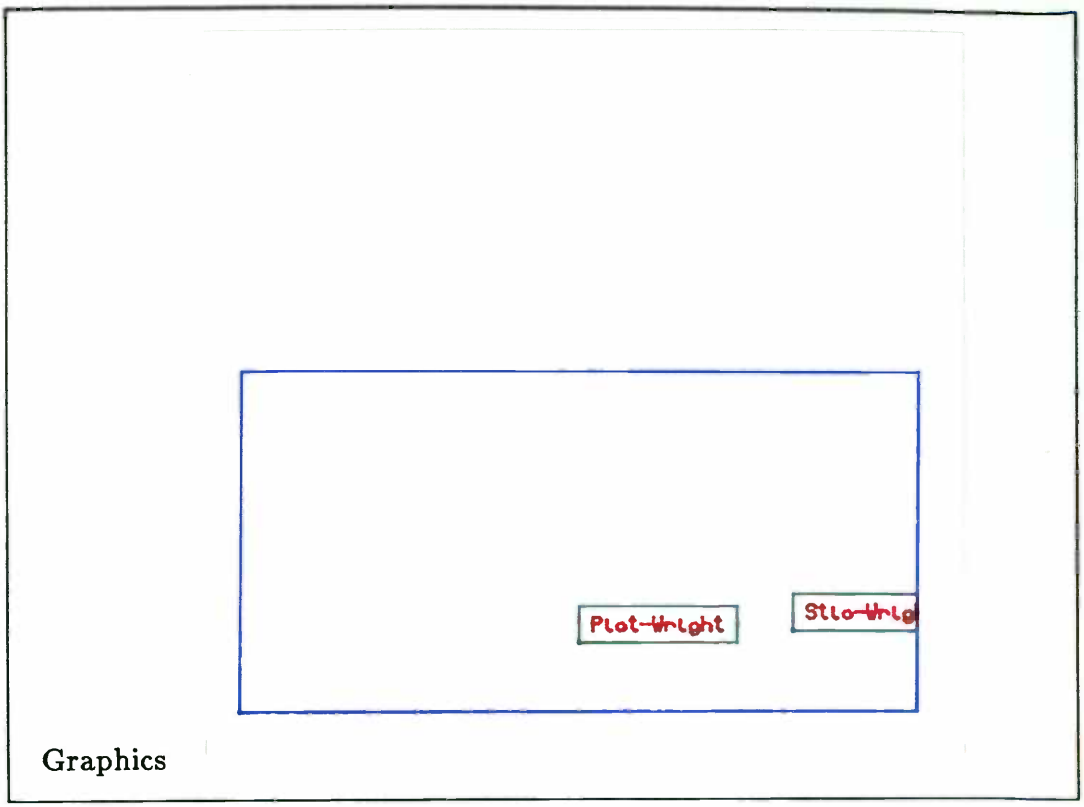
Figure 4–5 shows the new screen image after invoking a series of tree cursor movements (e.g. `MoveToSon`, `MoveToRightSibling`).

Now that the cursor is over the `List` non terminal covering the terminals `1, 2, 3` the command `ReplaceSubTree` can be invoked, and a set of new values entered via the Wright parser module. Here is the program script for the new `wrong` procedure call:

```
wrong (10000, 2000000, 30000, 4, 5, 6
       , 7, 8, 9, 10, 11, 12, 13, 114
       , 1555, 234234, 345, 123123,
       123132                        )
```

The `Wrong` call shows a weakness in the pretty-printing algorithm used by Wright; it would be more pleasing if the lines were broken after commas, and not numbers. The current algorithm, however, breaks a line at the first token which causes the line to be too large.

In order to remove the offending procedure call, the command `Delete SubTree` is invoked, an action described in Figure 4–6.



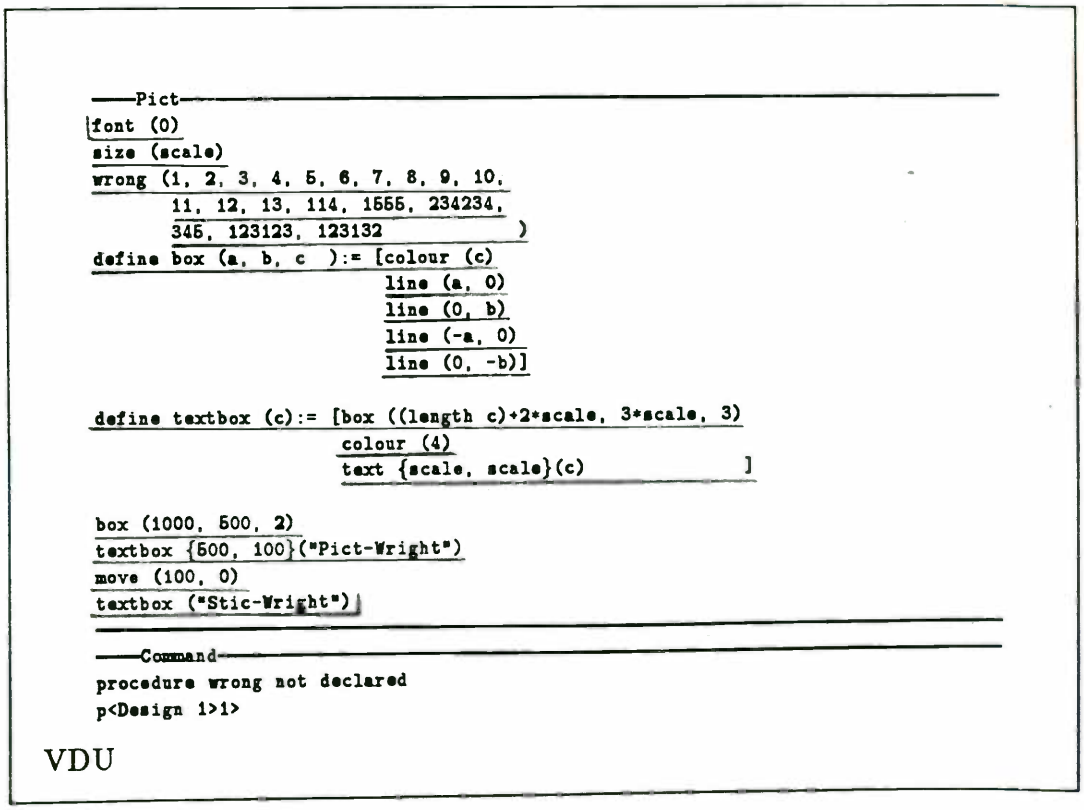
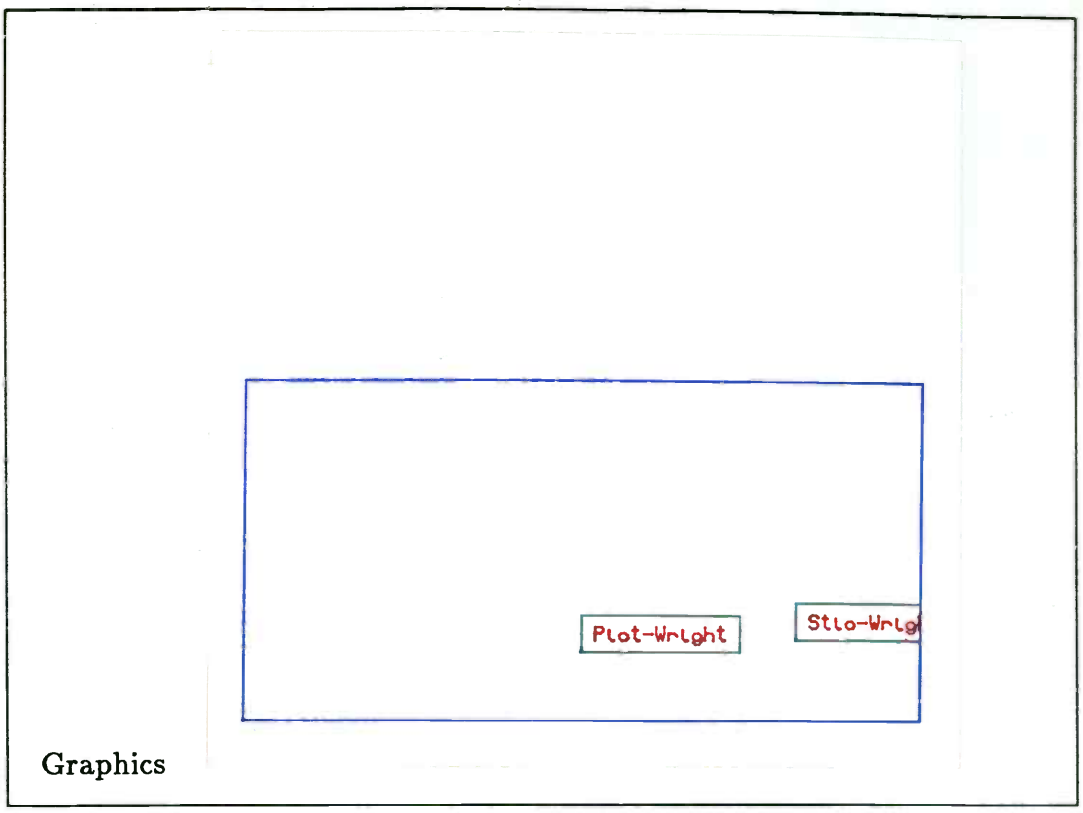


Figure 4–4: Editing with Pict-Wright




```
——Pict————————————————————————————————————————————————————————
font (0)
size (scale)
wrong (1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
       11, 12, 13, 114, 1555, 234234,
       345, 123123, 123132            )
define box (a, b, c  ):= [colour (c)
                          line (a, 0)
                          line (0, b)
                          line (-a, 0)
                          line (0, -b)]

define textbox (c):= [box ((length c)+2*scale, 3*scale, 3)
                      colour (4)
                      text {scale, scale}(c)              ]

box (1000, 500, 2)
textbox {500, 100}("Pict-Wright")
move (100, 0)
textbox ("Stic-Wright")
————————————————————————————————————————————————————————————————
——Command———————————————————————————————————————————————————————
p<List 1>1>
p<List 1>1>
```

VDU

Figure 4–5: A new cursor position

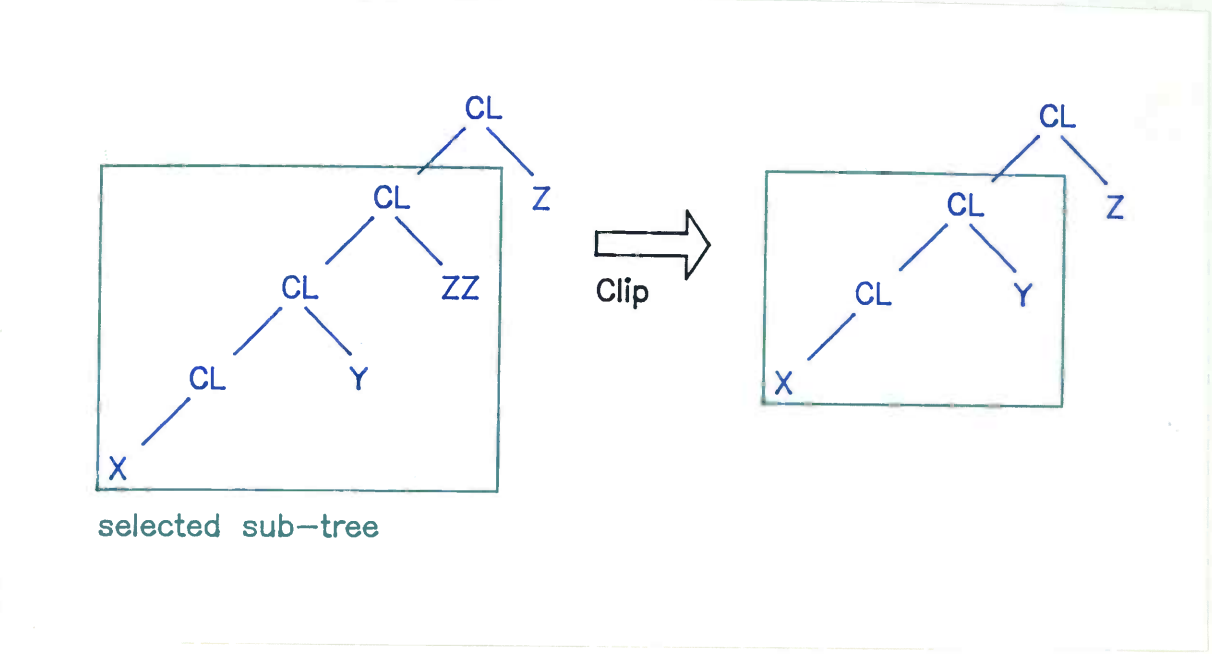
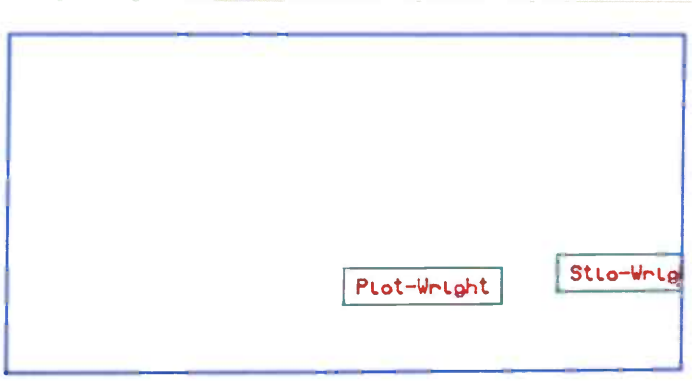


Figure 4–6: Deleting a sub-tree

After the previous sequence of editing commands the current graphics display image is:

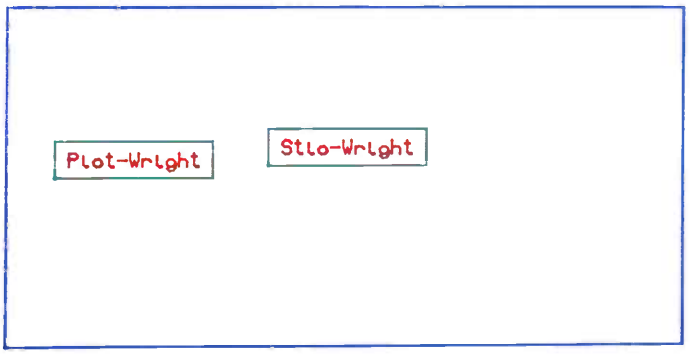


The editor cursor is now moved to the `List` non terminal of the first call of `textbox`. The command `ReplaceSubTree` is issued and the graphics *lexical macro* `Insert Coords` is invoked. This is a user supplied graphical interaction routine which invokes the graphics display's cursor, which can be moved using a mouse, and inserts the current screen position into the scanner's input buffer.

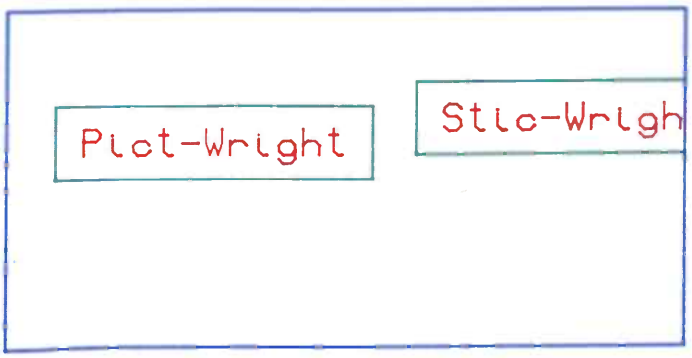
After inserting a new coordinate pair the text representation of the syntax tree becomes (on page 89):

```
define scale := 18

font (0)
size (scale)
define box (a, b, c ):= [colour (c)
                         line (a, 0)
                         line (0, b)
                         line (-a, 0)
                         line (0, -b)]

define textbox (c):= [box ((length c)+2*scale, 3*scale, 3)
                      colour (4)
                      text scale, scale(c)                ]

box (1000, 500, 2)
textbox {72, 249}("Pict-Wright")
move (100, 0)
textbox ("Stic-Wright")
```

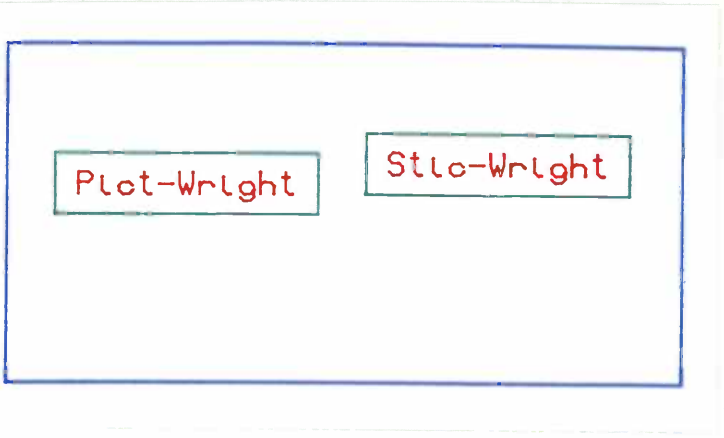
With these new coordinates the graphics display image is:



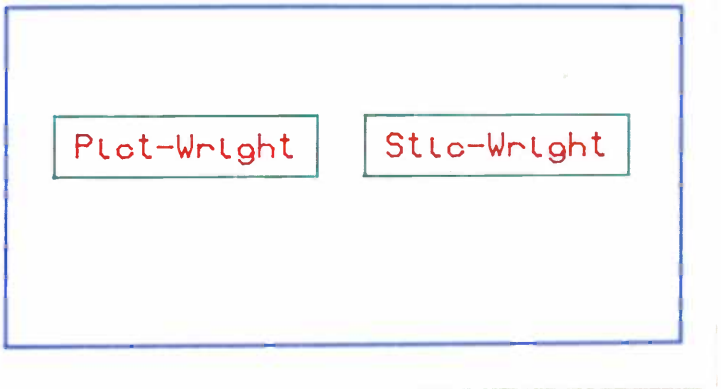
the picture is still a little small, and so the cursor is placed at the definition of `scale`, and a new value 36 is inserted:



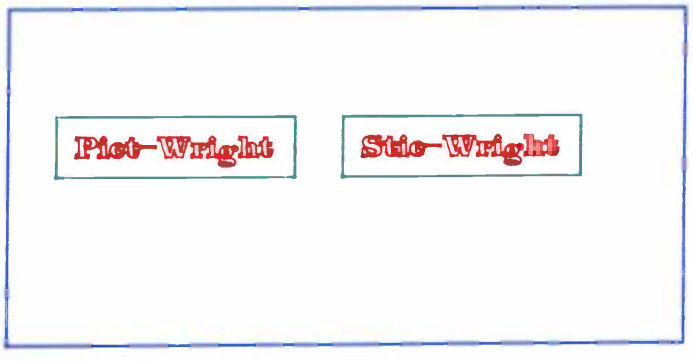
This is still too large, and so the value 30 is inserted:



The size now seems fine, so the cursor is now moved over `move (100, 0)` to correct the alignment between the two boxes. The value -30 is substituted in place of 0.



This positioning is fine, so move the cursor to `font(0)` and choose a more interesting font:



The program script now looks like this:

```
define scale := 30

font (62)
size (scale)
define box (a, b, c ):= [colour (c)
                         line (a, 0)
                         line (0, b)
                         line (-a, 0)
                         line (0, -b)]

define textbox (c):= [box ((length c)+2*scale, 3*scale, 3)
                      colour (4)
                      text scale, scale(c)                ]

box (1000, 500, 2)
textbox {72, 249}("Pict-Wright")
move (100, -30)
textbox ("Stic-Wright")
```

This kind of editing sequence is very easy to perform, and the evaluation time for each new screen image is negligible (although the current implementation is rather slow when using fonts other than the default, however, the default font can be used until the picture is nearly finished).

Pict-Wright is a small system with some inefficient implementation strategies, however, it performs well as an interactive environment for the production of simple text/line illustrations. The specification of the system by means of an attribute grammar proved to be a rewardingly straight-forward operation, and led to a very succinct implementation. The major work during the implementation of Pict-Wright was in refining the generic system modules (e.g. the editor commands and the interactive parser) since Pict-Wright was the first editor generated by Wright. The improvements made in the generic modules are, of course, passed on to any subsequently generated editor.

Chapter 5

Stick-Wright

5.1 Introduction

A major problem in the design of large integrated circuits is the assembly of circuit elements (e.g. RAMs, PLAs, adders, gates, pads etc.) in such a way as to minimise chip area which is devoted to wiring. Not only does wiring take up valuable space, it also significantly contributes to parasitic capacitances and resistances. Large routing channels can occur when an attempt is made to assemble modules together which have incompatible and fixed topologies.

A current trend in commercial IC development is the implementation of mainframe architectures in minimal chip-sets. The internal architecture of these systems has been heavily influenced by the designers' experience with previous technologies. At the start of a new technology it is expedient to draw on previous system design in order to produce working products quickly. The early LSI processors are an illustration of this phenomenon. The architecture of these chips (e.g. INTEL 4004, 8008 and 8080 families) resemble the single bus distributed system architecture employed by some of the PCB based processors of that time. With the next generation of LSI processor the bit-sliced architecture was adopted, bringing better performance. Present day partitioning of the IC design problem is still based on an effective method for assembling discrete devices on a PCB [79]. Chip assembly techniques based on the placement and routing of large and fixed cells (e.g. standard cells, PLAs, user leaf cells) tend to produce large areas of wiring channels. For better results cells must be designed to fit together, and library generated parts must be configurable to fit a variety

of contexts. The need to be fully aware of the physical design constraints of silicon right from the earliest stages of system design is succinctly described by Anceau, "The future of computer hardware must be imagined on silicon" [2].

Much research is currently going on into improving the physical design process. The tools and techniques being developed generally seem to apply in one of two directions; top-down or bottom-up. The top-down tools are concerned with floorplanning; placing the blocks at the top level in the structural hierarchy so as to best meet their connectivity and sometimes continuing this process down through the design hierarchy. Estimates, often relying on previous experience or guess work, have to made about block sizes and aspect ratios. These actual block dimensions are ultimately provided by the bottom up-tools; the symbolic compactors, cell generators and leaf cell editors. Interesting oscillations can occur between top-down and bottom-up design decisions. The most compact realisation of a particular cell may actually turn out to be more wasteful than one designed to be compatible with its neighbours. Similarly a change in the floorplan might make the compacted cell's topology more acceptable.

Physical design systems can also be categorised by two different approaches; design automation (DA) and computer aided design (CAD). Typical of the design automation approach are the floorplanning, placement and cell layout tools which use optimisation techniques (e.g. simulated annealing [23] [76]) or heuristics (e.g. as captured in an expert system [40]) to attempt solutions to problems for which tractable algorithms have not yet been found. Much success has been achieved in areas like routing (which generally is no longer trusted to human layout designers) and cell compaction, where polynomial algorithms have been found which give acceptable results (although full 2-D compaction has been shown to be NP-complete [78]). At the present time it is the tasks at the lower end of the physical design hierarchy that are best understood, e.g. routing and cell compaction. At the higher level of automatic cell generation (where the system is given the structure of the circuit, but not the planar embedding) research is not so far advanced, although the graph theoretical approach of Ng [63] shows some promise. The system described by Ng generates several circuit

topologies from an initial stick-diagram, choosing the one which proves to be the most compactable (although it may not necessarily be the best topology wrt. cell abutment). At the higher levels, the floor-planning tools introduced in Chapter 2, the automatic tools still have a long way to go.

In the CAD approach the primary design decisions are made by a human designer, with the computer aiding in the visualisation and verification of each design step. As tools ascend the layers of abstraction in the design hierarchy, the role of the human designer becomes more important, as the problems being solved become more open-ended. Complete design systems often make use of both CAD and DA approaches; the system described by Piguet [68] combines a manual floorplanning stage with an automatic cell layout stage.

Stick-Wright is a VLSI design tool for allowing VLSI circuits to be "imagined on silicon", and as such concentrates on the visual *exploration* and the *verification* aspects of the CAD approach. The major feature of the system is a symbolic layout language which has both textual and graphical representations. The ability to manipulate the design through both these mediums is the major contribution to design exploration. The enforcement of certain cell composition rules by the attribute grammar specification is the contributing factor to increased design verification.

Rem and Mead [70] suggested a set of design constraints for CMOS circuits which can be enforced through the syntactic and semantic rules specified for a design language. An attribute grammar provides an implementation method for just such a set of constraints, and Stick-Wright demonstrates the application of grammar rules to the type-checking of port compositions.

The system is intended to be used as a means for entering hierarchical floorplans of system components, with the lowest level cells being made up of wire and transistor primitives. Cells are constructed by the vertical or horizontal composition of sub-cells, forming rows and columns of abutted rectangles. This resulting *grid* is used to capture relative placement and to provide a framework for the port type-checking mechanism. As in similar grid based systems, the

tiling of cells does not necessarily convey information on actual sizes or aspect ratios.

The attribution rules in the AG given for Stick-Wright are for a *static* subset of the originally conceived system, and do not implement parameterisation in cell definitions. The presented implementation does, however, fully demonstrate AG specification techniques for the syntactic checking of cell compositions and the incremental construction of pictures. The syntactic definition of Stick-Wright, as contained in its AG, includes the parameterisation phrases of the extended version of Stick-Wright. Notational devices and implementation strategies for this extended system are presented in Section 5.6.

Stick-Wright is designed as the front-end to a cell-compaction chip-assembly tool. While the attributed parse tree maintained by the system contains all the information necessary to generate a data-structure suitable for these activities, the integration with further design stages was not a task undertaken for this thesis. The major concern of this thesis is the role of the front-end in fully exploiting the DA algorithms and techniques surveyed in Chapter 2.

Another objective of Stick-Wright is to demonstrate that the AG specification technique is effective in certain key areas, and that those designing production quality systems could benefit from adopting these techniques in some form (the problems of using current AG technology in production quality tools is discussed in the final chapter).

5.2 Lexical Definition

As was done for Pict-Wright, I begin the description of Stick-Wright by presenting its lexical analysis specification:

```
Lexical_definition Stic is

  Ranges
    @L is 'a' .. 'z' + 'A' .. 'Z';
    @N is '0' .. '9';
```

```
  @B is 0 .. 32;
  @S is 0 .. 127 - '"';
  @NotNL is 0 ..127 - 10;
end of ranges

macros
  #case is $$;
  #p is $($)*;
  #o is ($|)
end of macros

expressions

        #case;

        _cell    -> \cell;          _abut    -> \abut;
        _true    -> \true;          _false   -> \false;
        _b       -> \b;             _g       -> \g;
        _r       -> \r;             _bs      -> \bs;
        _gs      -> \gs;            _rs      -> \rs;
        _bc      -> \bc;            _gc      -> \gc;
        _rc      -> \rc;            _rbx     -> \rbx;
        _gbx     -> \gbx;           _pass    -> \pass;
        _enh     -> \enh;           _dep     -> \dep;
        _rbc     -> \rbc;           _gbc     -> \gbc;
        _rgcs    -> \rgcs;          _rgcc    -> \rgcc;
        _bt      -> \bt;            _gte     -> \gt;
        _rt      -> \rt;



        _equs    -> \=;             _ass     -> \:\=;
        _lb      -> \(;             _rb      -> \);
        _slb     -> \[;             _srb     -> \];
        _clb     -> \{;             _crb     -> \};
        _lt      -> \<;             _gt      -> \>;
        _arrow   -> \-\>;           _choice  -> \|;
        _tilda   -> \~;             _hat     -> \^;
        _comma   -> \, ;            _colon   -> \:;
        _semi    -> \;;             _stop    -> \.;
        _dots    -> \.\.;           _minus   -> \-;
        _plus    -> \+;             _times   -> \*;
        _divide  -> \/;             _and     -> \&;
        _or      -> \!;             _not     -> \\;

        _identifier  -> @L(@L|@N)*;
        _integer     -> @N@N* {was that a radix ?}
                        ( {yes} _(@N|@L) ! (@N|@L)* | {no} );

        _blank -> @B@B*;

        _comment -> \\\-@NotNL*\
```

```
  end of expressions

end of lexical_definition
```

The ASG statistics for this specification are:

```
ASG: parsing complete with no errors in 2030ms
ASG: DFA took 39388ms to build
ASG: DFA took 772ms to minimise
ASG:Compact: old size = 77*56 = 4312 entries
ASG:Compact: new size = 77*2 + 275*2 = 704 entries (38 misses)
             in 7806ms
```

The HARD statistics are:

```
HARD: scanner token _blank will be ignored by parser
HARD: scanner token _comment will be ignored by parser
HARD:Compact: old size = 77*127 = 9779 entries
HARD:Compact: new size = 77*2 + 814*2 = 1782 entries (100 misses)
              in 38148ms
```

5.3 Syntactic Definition

To simplify the explanation of the attribution rules, I give a brief description of the underlying syntax and an informal introduction to the semantics. The grammar presented here also includes the parameterisation phrases discussed in Section 5.6.

```
Grammar Stic is

  Productions

    Design -> CellList Ident;

    CellList -> CellList Cell | Cell

    Cell   -> _cell _identifier Params PortSpec _equs Abut _stop;
```

A Stick-Wright design consists of one or more `Cell` definitions followed by an `Ident` (which can be a primitive `Cell` or a call to a defined `Cell`).

```
Params   -> _clb IList _crb |;

IList -> IList _comma _identifier | _identifier;
```

A cell may have a list of parameters.

```
PortSpec -> _lb OPorts _rb;

OPorts -> OList _semi OList _semi OList _semi OList |;

OList -> List |;

List -> List _comma Id | Id;

Id -> _lb List _rb QualS OIter |
      _identifier QualS OIter ;

QualS -> QualS _colon ODir _identifier |;

ODir -> _gt | _lt;
```

Each cell has a list of ports corresponding to the West, North, East and South cell boundaries:

```
Cell Tally {n}(vdd:d, gnd:d[1 ..  n];
               ;
               z:d[n+1 ..  n];
               (X:m, gnd:d, Xbar:m)[n ..  1]) = ...
```

In this example the West `OList` contains a single `vdd` port on layer `d` (diffusion) and a vector of `gnd` ports. The North side of the cell has no ports and the South side has a vector of three ports grouped together (i.e. all three are iterated together). All ports are ordered in a clockwise direction, hence the vectors on the West side count up, and the vectors on the East count down. Figure 5–1 shows the corresponding picture for this group of port declarations.

```
Abut -> _abut AbutBlock;

AbutBlock -> AbutBlock _semi Row | Row;

Row -> Row _comma Item | Item;
```

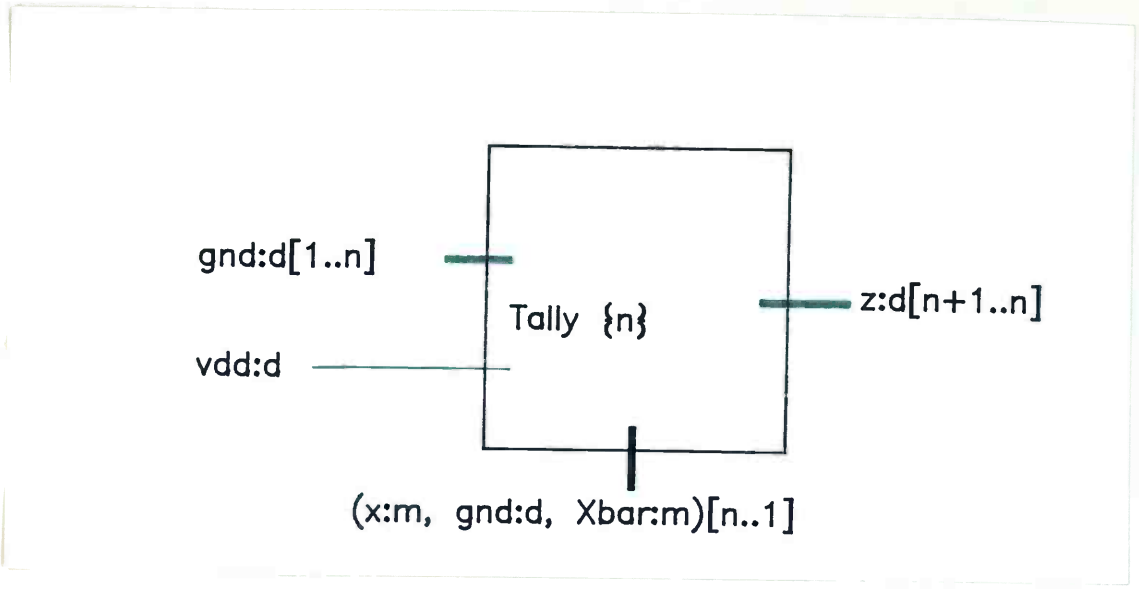


Figure 5–1: Port Exterior for `Tally {n}`

```
VRow -> VRow _comma Item | Item;

Item -> Ident Syms OIter;
```

Cells are formed by the abutment of other composition cells, or primitive cells. The `AbutBlock` is formed from the vertical composition of `Rows`, which in turn are composed of the horizontal abutment of `Items`. An `Item` consists of a reference to a more primitive cell structure, `Ident`, an optional series of geometrical transformations, `Syms`, and an optional iteration operator, `OIter`.

```
Ident -> _lb Row _rb        |
         _lt VRow _gt       |
         _slb Cond _srb     |
         _identifier  OPar  |
         _b    | _g    | _r    | _bs   | _gs   | _rs   | _bc   |
         _gc   | _rc   | _rbx  | _gbx  | _pass | _enh  | _dep  |
         _bt   | _gte  | _rt   | _rbc  | _gbc  | _rgcs | _rgcc |;

OPar  -> _clb Pars _crb |;

Pars  -> Pars _comma Expression |
         Expression;
```

The `Ident` cell abutment primitive can itself be a nested `Row`; round brackets `( ...)` indicates horizontal abutment, angle brackets `< ...>` indicates vertical

abutment. There is also a conditional form Cond indicated by square [...] brackets. The fourth Ident is a cell call, the rest are primitive cells (wires, contacts and transistors). Figure 5–2 shows the primitive cells. The non terminal OPar is the optional parameter list for a cell call.

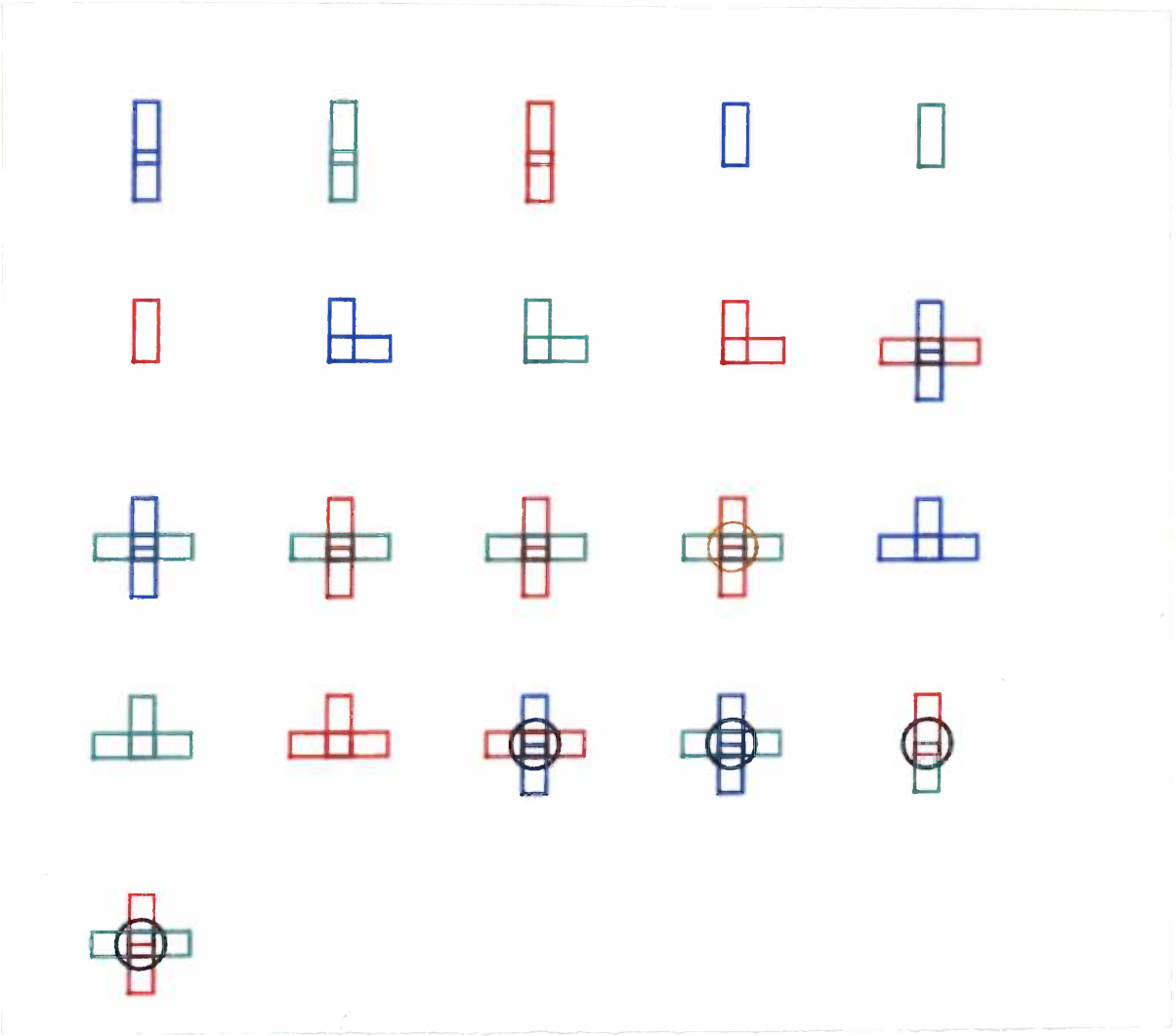


Figure 5–2: Stick-Wright's Primitive Cells

```
OIter -> _slb Iter _srb |;

Iter  -> _identifier _ass Range |
         Range;

Cond -> Condition _arrow Item OBar;

Syms -> Syms Sym |;
```

```
Sym   -> _tilda | _hat;

Range -> Expression _dots Expression |
         Expression;

OBar  -> _choice Item |;
```

An `Iter` phrase contains an iteration specification, which can also declare an iteration variable, for use as a parameter to the cell being iterated. A range can be between two expressions, e.g. `2*k .. n`, while a single expression indicates a ranging starting from 1, with the expression denoting the final position e.g. `1 n-1`.

A `Cond` statement will choose the first `Item` after the `_arrow` if `Cond` is *true*, otherwise the `Item` contained in `OBar` (possibly null).

The `Sym _tilda` (~) is a post-fix operator indicating reflection in the Y-axis, the `Sym _hat` (^) indicates a clockwise 90° rotation.

```
Expression -> _minus Expression             |
              _lb Expression _rb            |
              Expression _times Expression  |
              Expression _divide Expression |
              Expression _plus Expression   |
              Expression _minus Expression  |
              _identifier                   |
              _integer;

Condition -> _lb Condition _rb            |
             _not Condition               |
             Condition _and Condition     |
             Condition _or Condition      |
             Expression _equs Expression  |
             Expression _lt Expression    |
             Expression _gt Expression    |
             _true                        |
             _false;

End of Productions

Priorities (_times, _divide) (_plus, _minus);

End of Grammar
```

The grammar ends with a set of integer expression and condition phrases. Figure 5–3 on page 103 shows a complete Stick-Wright script, with its corresponding picture.

The following statistics are taken from diagnostics issued by the Wright parsing building module:

```
WRIGHT building PDA ......
WRIGHT PDA generated in 48858ms
WRIGHT lookaheads generated in 133260ms
WRIGHT s/r :  [  3, _slb] <state #  7, Ident> =>s          (i)
WRIGHT s/r :  [ 62, _times] <state # 70, Expression> =>s   (ii)
WRIGHT s/r :  [ 63, _and] <state # 73, Condition> rL       (iii)
WRIGHT tables took 63376ms to fill
WRIGHT old size = 151*54 = 8154 entries
WRIGHT new size = 151*2 + 1008*2 = 2318 entries (168 misses)
       in 37376ms
WRIGHT old size = 151*29 = 4379 entries
WRIGHT new size = 151*2 + 121*2 = 544 entries (21 misses)
       in 6057ms
WRIGHT tables printed in 112749ms
```

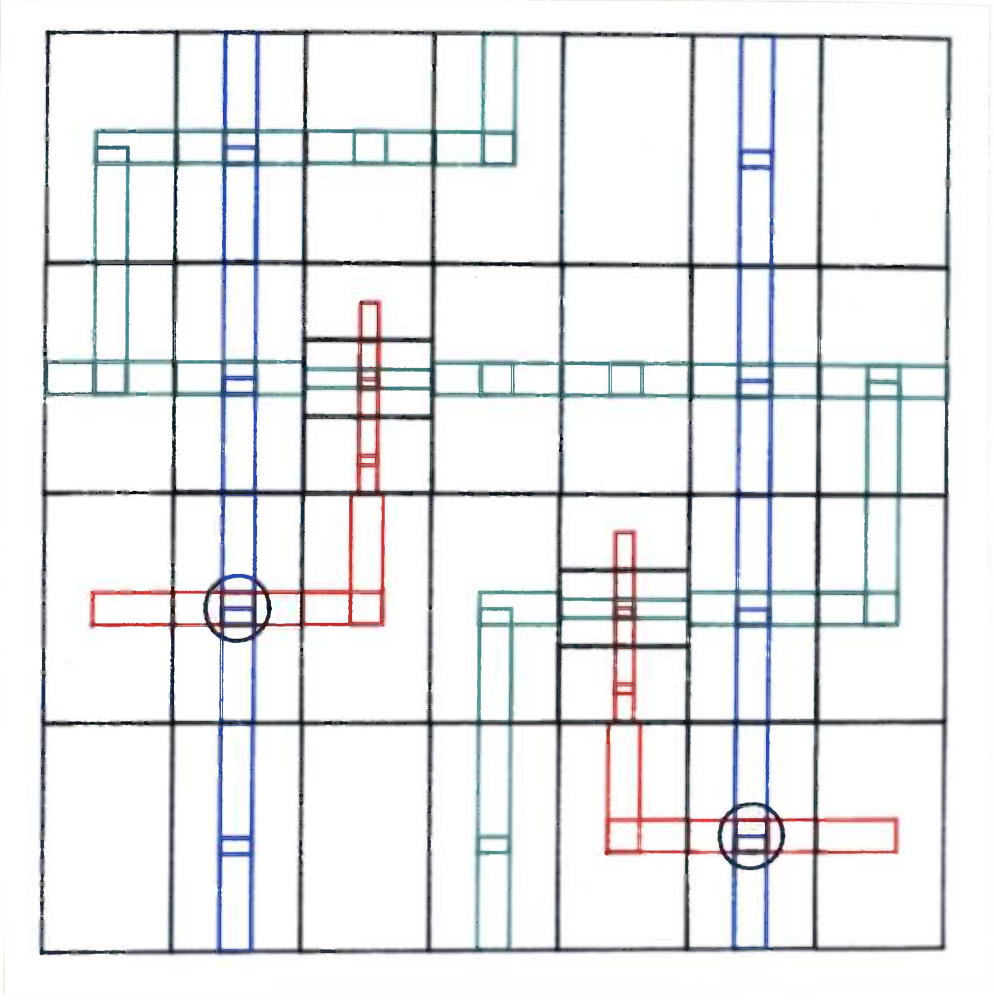
The lines marked `(i)`, `(ii)`, `(iii)` are some of the *shift/reduce* errors detected by the system:

`(i)` This ambiguity arises from the fact that the syntax for a conditional `Ident` is the same as an iterated null statement in an `Item` phrase. This ambiguity was left as it was because the default action of the parser is to *shift* on the `_slb`, thus making the assumption that the conditional `Ident` is following, however, the iterated null `Ident` can be achieved using the syntax `()[iter_variable]`.

`(ii)` This kind of ambiguity is resolved by the precedence rules given at the end of the grammar.

`(iii)` This kind of ambiguity is resolved by taking the default action of assuming *left-associativity* of operators.

The complete processing of the AG by Wright takes just over 5 minutes.



```
Cell TallyUnit (in1:d;
                Xbar:m, out2:d, X:m;
                out1:d;
                X:m, in2:d, Xbar:m ) =
  Abut gc^, gbx, g^, gc^^^, , b, ;
       gt, gbx, <rs^^, pass, r>, g^, g^, gbx, gt^^;
       rs^, rbc, rc^^^, gc^, <rs^^, pass, r>, gbx, gc^^^;
       , b, , g, rc, rbc, rs^^^

TallyUnit
```

Figure 5–3: A TallyUnit

5.4 Semantic Definition

5.4.1 The Attributes

There are nine synthesised attributes (refer to Appendix C for their respective non terminals):

box This is the same pretty-printing attribute described in the last chapter.

ports This attribute contains the current *port exterior* of its non terminal, and is used in the verification of cell compositions:

```
%record %format Port Block Fm (%record(Port List fm)%name N, E,
                                                        S, W,
                               %integer mir,
                               %string(*)%name id)
```

A `Port Block` has four sides of ports, may be mirrored and is given a `name`. A `Port List`:

```
%record %format Port List Fm (%record(Port fm)%name P,
                              %record(Port List fm)%name next)
```

is a list of `Ports`:

```
%record %format Port fm (%string(*)%name id,
                         %record(Constraint List fm)%name Cs)
```

which have a name, and a list of `Constraints`:

```
%record %format Constraint List Fm (
                         %integer dir,
                         %string(*)%name id,
                         %record(Constraint List fm)%name next)
```

A `Constraint` can impose a *signal direction*, `dir`, and a signal name, `id`.

`def` The `def` attribute is similar to the symbol table attribute in Pict-Wright, storing bindings of cell names to their definitions.

`x` This attribute stores the number of `Item` non terminals in a `Row` or `VRow` cell abutment phrase.

`y` This attribute stores the number of `Row` non terminals in a `AbutBlock` cell abutment phrase, or the number of `Item` non terminals in a `VRow` cell abutment phrase.

`mir, rot` Attributes which indicate the geometrical transformations to be performed for a `Ident` phrase in a `Item` phrase. (`mir` indicates *mirror* in the Y-axis, `rot` indicates a 90° clockwise rotation.

`val` The value of an expression. In the present implementation only constant expressions (i.e. not containing identifier references) are meaningful.

`bool` The value of a boolean expression..

There are three inherited attributes:

`env` The symbol table containing bindings of cell names to cell definitions. Again, this is implemented using the method presented in Pict-Wright, however, some of the fields are different:

```
%record %format env fm (%string(*)%name id,
                        %integer addr,
                        %integer x, y, ports,
                        %record(env fm)%name next)
```

`addr` The address of the cell definition in the attributed parse tree.

`x, y` The `x,y` attributes synthesised for that definition.

`ports` The port exterior synthesised for that definition.

`origin` This is the attribute crucial to the implementation of incremental picture drawing, and contains a *geometrical transformation matrix* [61] [22] for the current non terminal:

```
%record %format TRANS FM (%real %array A (0:8))
```

where **A** represents the 3×3 transformation matrix. This matrix store the current composition of translations, rotations, scalings and mirrorings which will transform points drawn for the current non terminal onto an appropriate region of the graphics display. The basic transformations are obtained from the following formulations:

Translation by $T(x, y)$:

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix}$$

Rotation by θ:

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Scaling by $S(x, y)$:

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Mirroring in Y:

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

compositions of transformations are achieved by matrix multiplication.

dir This attribute informs an `Item` non terminal whether it is being horizontally composed in a `Row` phrase, or vertically composed in a `VRow` phrase.

In Stick-Wright there are no active drawing attributes, i.e. pictures are not drawn as the result of attribute evaluation, rather, a separate tree-walking procedure generates the picture in a depth-first traversal of the attributed syntax tree. The evaluation of the `origin` attribute decorates the tree with all the positioning and scaling information needed for displaying the graphical representation of the design (this is similar to the way the `box` attribute provides the pretty-printer with its data). The evaluation of this graphics data-structure enjoys the benefits of the incremental evaluation scheme, and also provides the means for directly displaying any part of the design, as represented in any arbitrary sub-tree.

5.4.2 Semantic Functions

The semantic functions for pretty-printing and symbol table management are as presented in Pict-Wright, with the exception that Stick-Wright does not have identifier/expression-value bindings, and so the symbol table has only to deal with cell names. The major new attribution rules in Stick-Wright are those for the syntactic checking of port-compositions through the attribute `ports`, and also those for determining the transformation attribute `origin`[1]:

```
Design -> CellList Ident
          <ports$0  = ports$2>          (1)
          <origin$2 = top origin>       (2)
```

Attribution No. (1) triggers the evaluation of the port exterior for the whole design, determined by `Ident`, which in turn triggers the inherited attribution of No. (2), which gives the top level `Ident` the unity transformation matrix, thus providing it with the whole of the graphics display (all drawing commands assume that they have the whole display available, but the scaling component of their `origin` matrix will make the corresponding `Ident` picture the appropriate size).

[1] `origin` is an extension of the coordinate attribute of Pict-Wright to include scaling, rotation and mirroring information

```
CellList -> CellList Cell
             <env$2 = def$1>                                    (3)
             <def$0 = add def(def$1, def$2)>                    (4)
            Cell
             <def$0 = def$1>                                    (5)

Cell   -> _cell _identifier Params PortSpec _equs Abut _stop
           <def$0 = do binding( x$6, y$6, ports$6)>             (6)
           <env$6 = env$0>                                      (7)
```

Cell identifier/definition bindings are made in the same manner as in Pict-Wright, with successive cell definitions inheriting the accumulated symbol table (3). Recursive cell specifications are prevented by a cell's name not being in the symbol table inherited by the cell's Abut non terminal (7). The identifier binding operation in (6) triggers the evaluation of the cell's ports attribute (which will also later trigger the evaluation of its origin attribute).

Phrases Params ... ODir are part of Extended Stick-Wright (Section 5.6) and only have pretty-printing attributes.

```
Abut -> _abut AbutBlock                          (8)
         <ports$0 = ports$2>                     (9)
         <x$0 = x$2> <y$0 = y$2 >                (10, 11)
         <origin$2 = new origin(x$2, y$2)>       (12)
```

The x and y attribute for a Abut non terminal (obtained by copy rules (10, 11)) give the dimensions for the current cell's *tiling*, i.e. each composition cell is made up of the regular composition of rectangular blocks in the X and Y directions. The figure on page 103 shows a composition cell consisting of a 7×4 x/y tiling grid. The origin for the AbutBlock non terminal is constructed with x and y scaling factors which will cause its constituent parts to be scaled down to fit onto the tiling grid.

```
AbutBlock -> AbutBlock _semi Row
              <ports$0 = do Abut compose(ports$1, ports$3, y$1)> (13)
              <x$0 = check length(x$1, x$3)>                    (14)
              <y$0 = y$1 + 1>                                   (15)
              <origin$3 = do origin(origin$0, 0, y$1)>          (16)
              <origin$1 = origin$0> |                           (17)
             Row
```

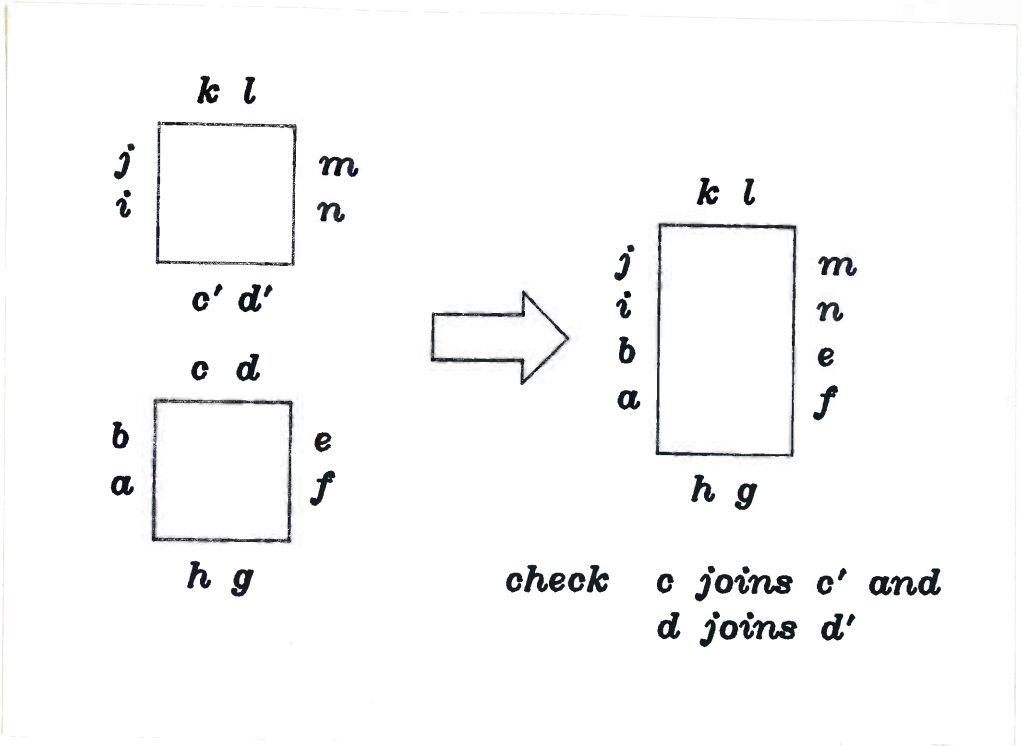


Figure 5–4: Vertical Port Composition

```
<origin$1 = origin$0>                                    (18)
<ports$0 = ports$1>                                      (19)
<x$0 = x$1> <y$0 = 1>                                 (20,21)
```

The AbutBlock phrase of Stick-Wight causes the vertical composition of the Row phrase. When composing two rows of ports the attribution rule do Abut compose (13) checks that the South port-list of the top block of ports ports$1 *matches* with the North port-List of the bottom block of ports ports$3. The port-block returned by this function has the North side of the top port-block, the South side of the bottom port-block and the concatenation of the ports on the East and West sides. Figure 5–4 illustrates this attribute evaluation. Matching, in the context of port-type checking, is defined to mean *having the same port-constraint-lists.* A more elaborate definition of what constitutes a valid match is given in Extended Stick-Wright.

Attributions Nos. (14,15,20,21) construct the tiling attributes x and y, attribution No. (14) also checks that the rows in an AbutBlock are the same length (and signals an attribution error if they are not the same).

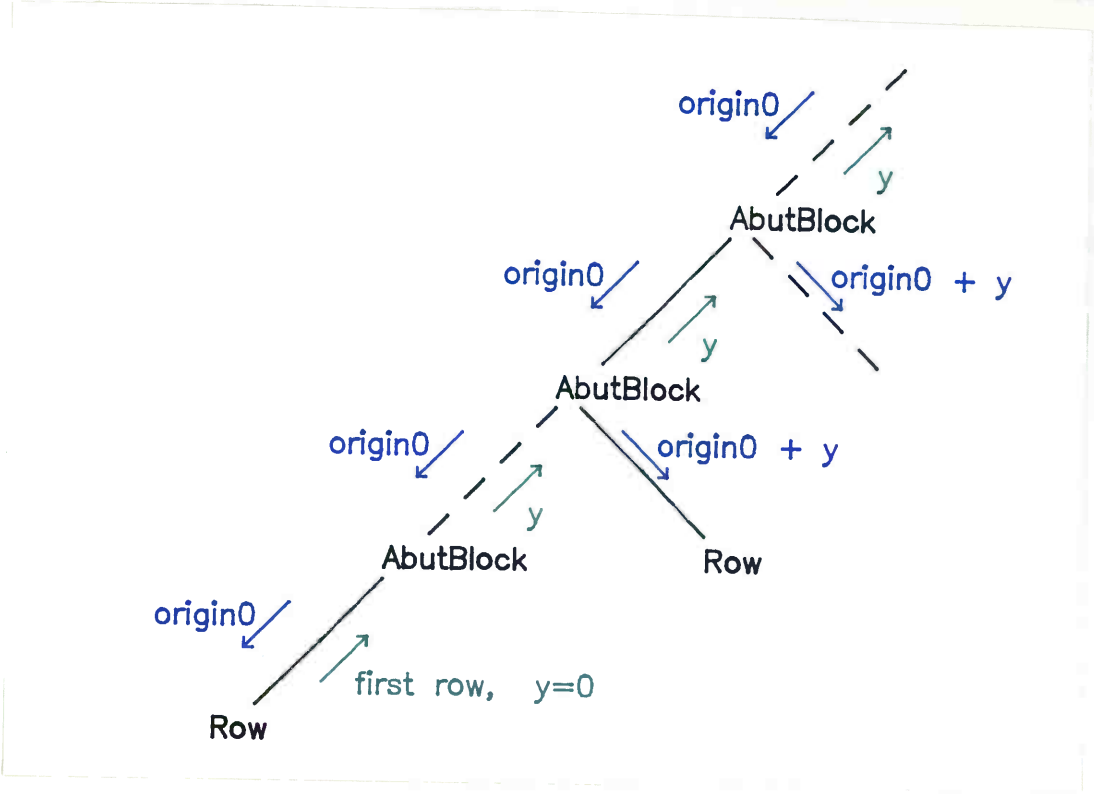


Figure 5–5: Row Translation Calculation

The `origin` attributions, Nos. `(16,17,18)` pass on the scaling information determined for the current tiling, but now add on the appropriate translation transformation, in order to position each row correctly. Attribution No. `(18)` has no translation, since the `origin` given from the `Abut` phrase is also the origin of the top row. Attribution No. `(16)` gives the translation from the top of the tiling grid to the current row being added (a distance which is immediately available from the tiling attribute y). Figure 5–5 illustrates this calculation of translations.

```
Row -> Row _comma Item
        <ports$0 = do Row compose(ports$1, ports$3, x$1)>   (22)
        <origin$1 = origin$0>                               (23)
        <origin$3 = do origin(origin$0, x$1, 0)>            (24)
        <dir$3 = 0>                                         (25)
        <x$0 = x$1 + x$3> |                                 (26)
       Item
        <ports$0 = ports$1>                                 (27)
        <x$0 = x$1>                                         (28)
        <origin$1 = origin$0>                               (29)
        <dir$1 = 0>                                         (30)
```

The attribution rules for the horizontal composition of `Item` non terminals in a `Row` phrase follows a similar pattern to the vertical composition of rows. Figure 5–6 illustrates the horizontal port composition attribution of (22). The translation composed into the current `origin` attribute in (24) causes the correct positioning of the `Item` within the `Abut` tiling. The `dir` attributions (25,30) pass down the information that the current `Item` is undergoing a horizontal composition.

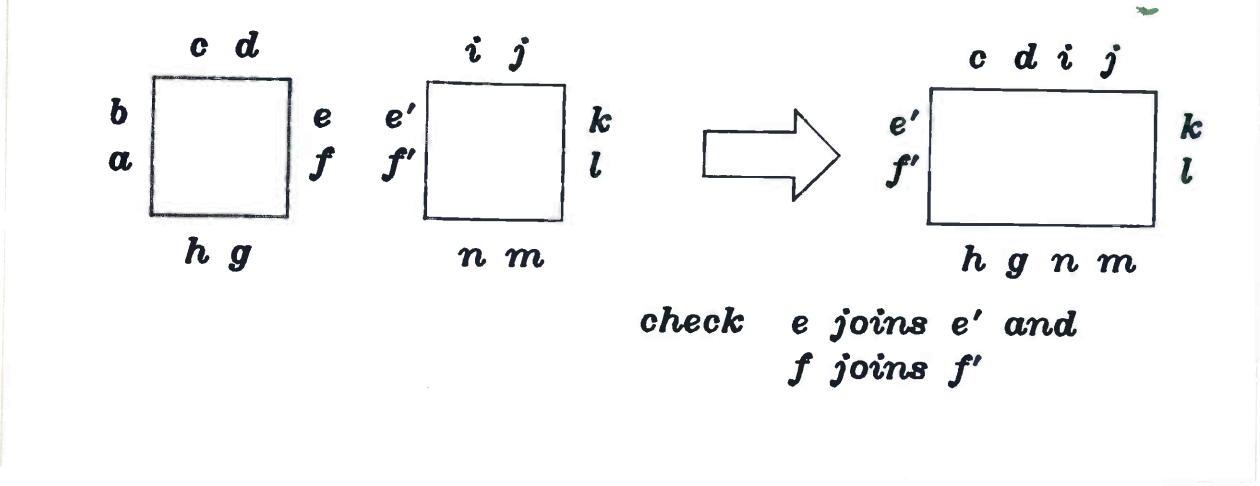


Figure 5–6: Horizontal Port Composition

```
VRow -> VRow _comma Item
        <ports$0 = do Abut compose(ports$1, ports$3, y$1)>   (31)
        <origin$1 = origin$0>                                (32)
        <origin$3 = do origin(origin$0, 0, y$1)>             (33)
        <dir$3 = 1>                                          (34)
        <y$0 = y$1 + x$3> |                                  (35)
       Item
        <origin$1 = origin$0>                                (36)
        <y$0 = x$1>                                          (37)
        <ports$0 = ports$1>                                  (38)
        <dir$1 = 1>                                          (39)
```

The VRow phrase follows the same pattern as the AbutBlock phrase. The dir attributions (34,39) pass down the information that the Item is undergoing a vertical composition.

```
Item -> Ident Syms OIter
        <x$0 = x$3>                                              (40)
        <ports$0 = do port trans(ports$1, mir$2, rot$2,
                                 x$3, dir$0)>                    (41)
        <origin$1 = do transforms(origin$0, mir$2, rot$2)>       (42)
```

An Item tile can take up several x-positions if it has a non-empty OIter phrase (40). The presence of geometric transformations in the form of a non-empty Syms phrase causes appropriate changes to the port exterior attribute ports (41), shown in Figure 5–7, and appropriate additions to the accumulated transformation matrix (42), shown in in Figure 5–8.

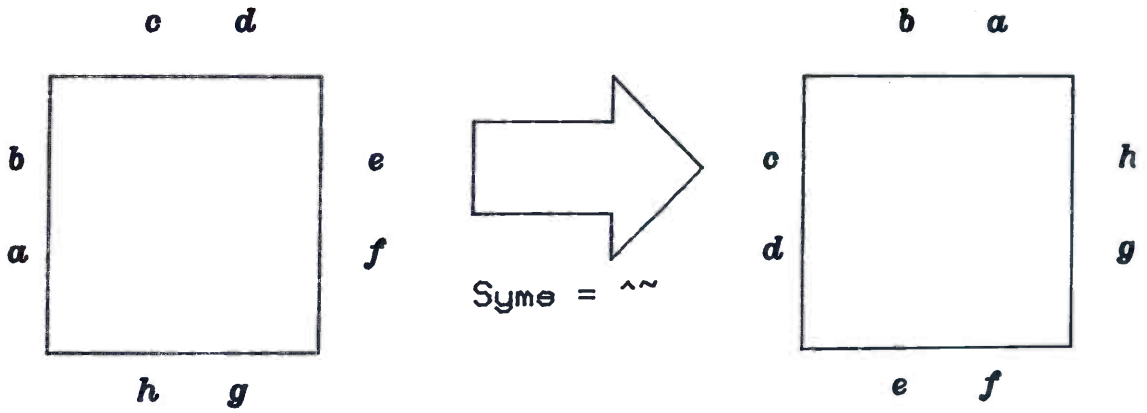


Figure 5–7: Transformations to Ports Exterior

```
Ident -> _lb Row _rb
         <origin$2 = do hor trans(origin$0, x$2)>        (43)
         <ports$0 = ports$2> |                           (44)
         _lt VRow _gt
         <ports$0 = ports$2>                             (45)
```

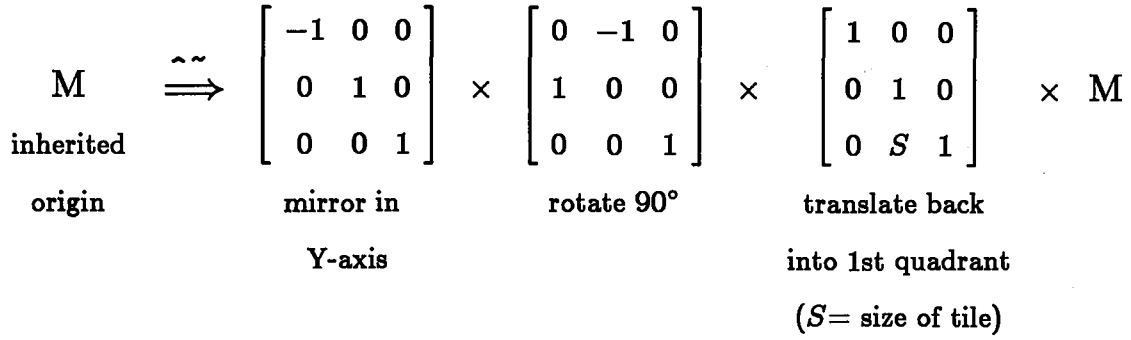


Figure 5–8: Mirroring and Rotation (^~)

```
 <origin$2 = do ver trans(origin$0, y$2)> |      (46)
_slb Cond _srb
 <ports$0 = ports$2>                             (47)
 <origin$2 = origin$0> |                         (48)
_identifier OPar
 <ports$0 = do call( env$0, origin$0)> |         (49)
_b
 <ports$0 = do b(origin$0)>                      (50)
```

The grouping of the Ident phrase into bracketed compositions of horizontally or vertically composed rows (43,44,45,46) involves the adding in of a further scaling transform to the inherited origin attribute. The conditional cell evaluation phrase is only really useful in Extended Stick-Wright, but a limited use is presented in the next section.

Attribution No. (49) performs an identifier lookup in the current symbol table (refer to the complete AG, Appendix C, to see the copy rules for the attribute env). The entry in the symbol table (if found, otherwise an attribution error is signalled) has as one of its fields the exterior port appearance of the called cell. This is then passed up the tree for use in cell composition checking.

Attribution (50) synthesises the port appearance of the primitive _b, a blue wire.

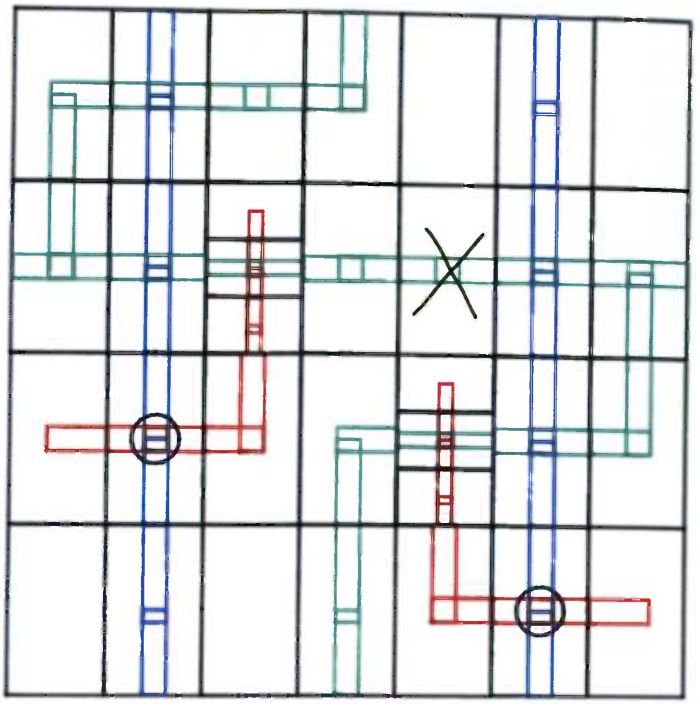
The attributions for the rest of the grammar are straight-forward copy rules, expression evaluations and condition evaluations. At present only constant expressions are permitted, and so expressions can be evaluated directly on the tree as synthesised attributes.

5.5 The Editor in Operation

I begin the presentation of the operational aspects of Stick-Wright by returning to the `TallyUnit` introduced earlier (Figure 5–3) and showing some editing actions being performed with it. Figure 5–9 shows the system configuration, with the `TallyUnit` as the currently selected cell.

The graphics cursor (the cross) in Figure 5–9 has been invoked by the major graphics interaction command of Stick-Wright, `SelectTile`, Figure 5–10 shows the system configuration after clicking the mouse at the position selected in Figure 5–9. `SelectTile` converts the window-coordinates provided by the mouse into *tile-coordinates*. The tile-coordinate-system has its origin at the bottom-right, so the coordinate corresponding to the tile selected in in Figure 5–9 is $(3,3)$, the lowest `rbc` is $(2,1)$ and the other `rbc` is $(6,2)$. These coordinates are is used to drive tree traversals on the `Abut` phrase in the `TallyUnit` parse tree; the new text cursor position is found by descending down the number of `AbutBlock` phrases given by the y coordinate, and then down the number of `Row` phrases given by the x coordinate. Figure 5–11 shows the moves taken to arrive at the second `g^` of the second `AbutBlock` from the top (which is the third `AbutBlock` down in the `Abut` phrase).

Relating device coordinates to particular entities in a graphics data-structure (an activity known as *hit detection*) is significantly simplified when the graphics screen has been segmented into non-overlapping regions, as in the tiling of Stick-Wright. In a data-structure without the tiling property, e.g. Pict-Wright, objects relating to points provided by a cursor position can be found by traversing the data-structure, searching for hits within a small area around this point. The



Graphics

```
----Stic----------------------------------------------------------------
Cell TallyUnit (in1:d;
                Xbar:m, out2:d, X:m;
                out1:d;
                X:m, in2:d, Xbar:m  ) =
  Abut gc^, gbx, g^, gc^^^, , b, ;
       gt, gbx, <rs^^, pass, r>, g^, g^, gbx, gt^^;
       rs^, rbc, rc^^^, gc^, <rs^^, pass, r>, gbx, gc^^^;
       , b, , g, rc, rbc, rs^^^                                    .

Cell routeS (;
             ;
             ;
             ) = Abut , b, , , , b, ;
                      g^, gbx, g^, g^, g^, gbx, g^;
                      , b, , , , b, ;
                      , b, , gs^^, , b,                          .

----Command-------------------------------------------------------------
p<CellList   2>>
p<Cell   1>>
```

VDU

Figure 5–9: The Stick-Wright Editor

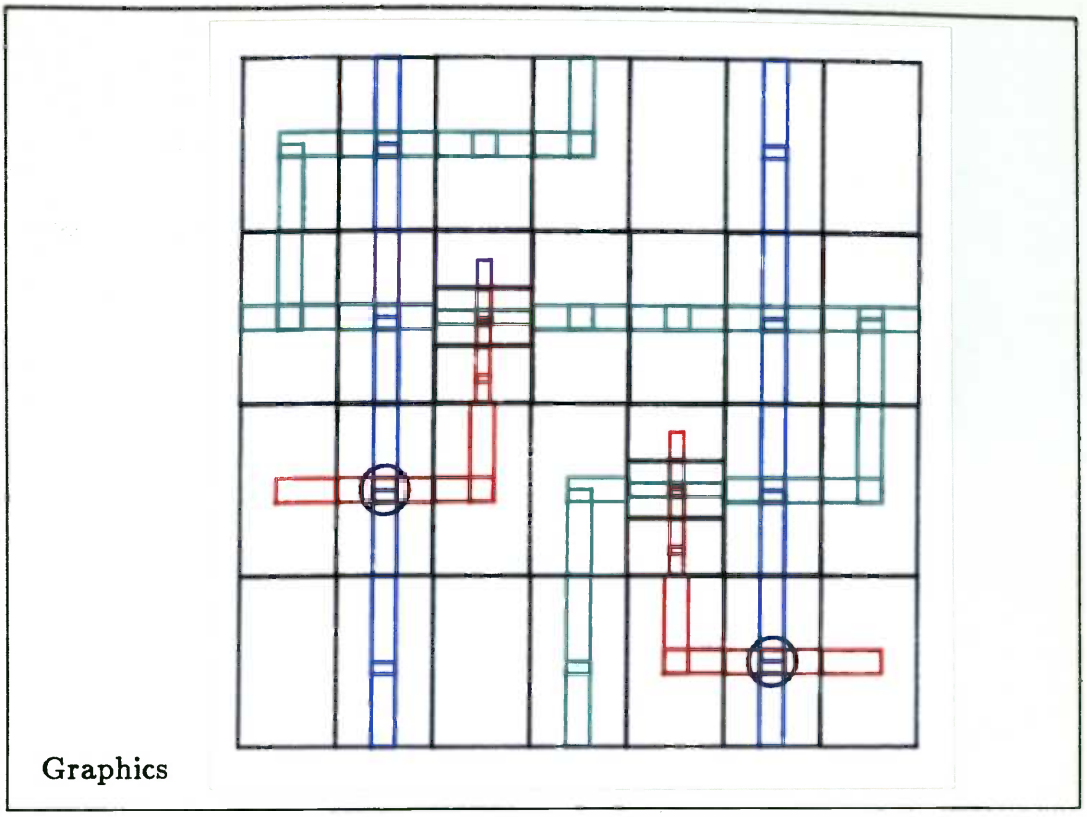


Figure 5–10: A new cursor position in `TallyUnit`

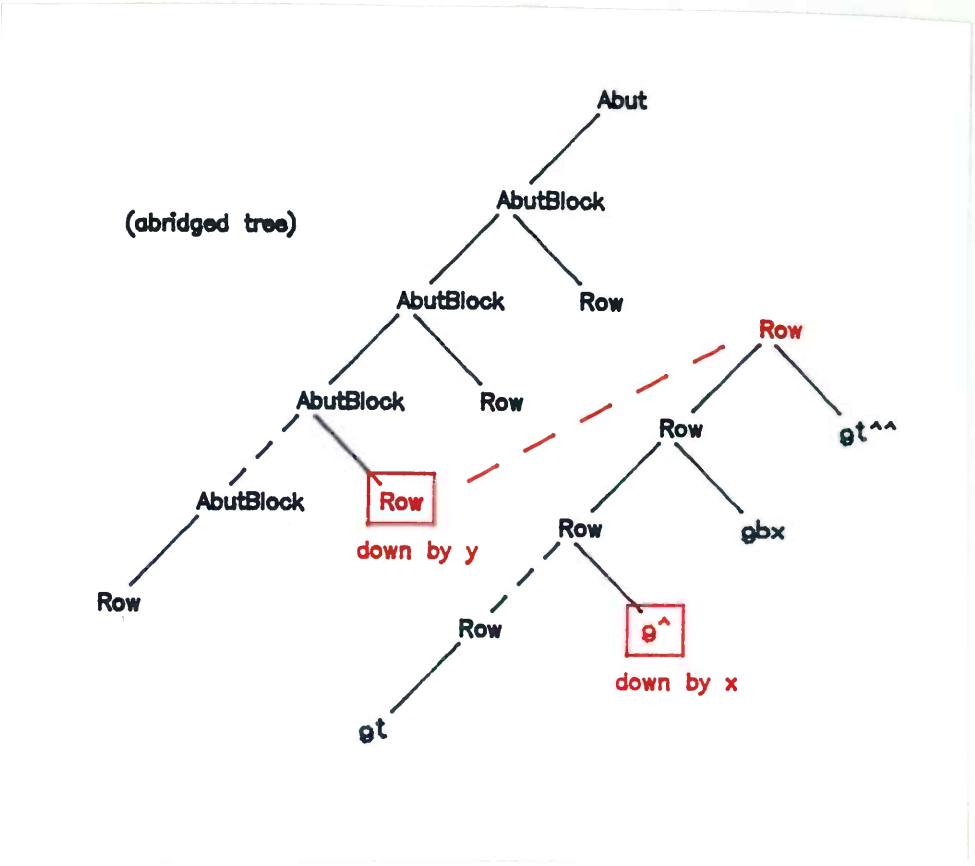


Figure 5–11: Moving to a new tile

Wright generated editors have an added advantage that this traversal/evaluation can be restricted to only those parts of the tree whose attributes have been altered.

Now that this primitive cell has been selected, the port type-checking attribution rules can be exercised by the insertion of an incorrect wire. The command `ReplaceSubTree` is given at this new position, and the `g^` is replaced by the primitive `r` (see Figure 5–12). Only the new tile needs to be drawn, the correct scaling and positioning being determined from the tile's `origin` attribute. This replacement causes the following attribution errors to be signalled:

```
| ports clash 3|g1 and r[0]
| port clash 4|r[0] and gbx
^ port clash 6|blank[3] and 6|gte2[4]
^ port clash 1^6|gte2[5] and 6|gc3[4]
```

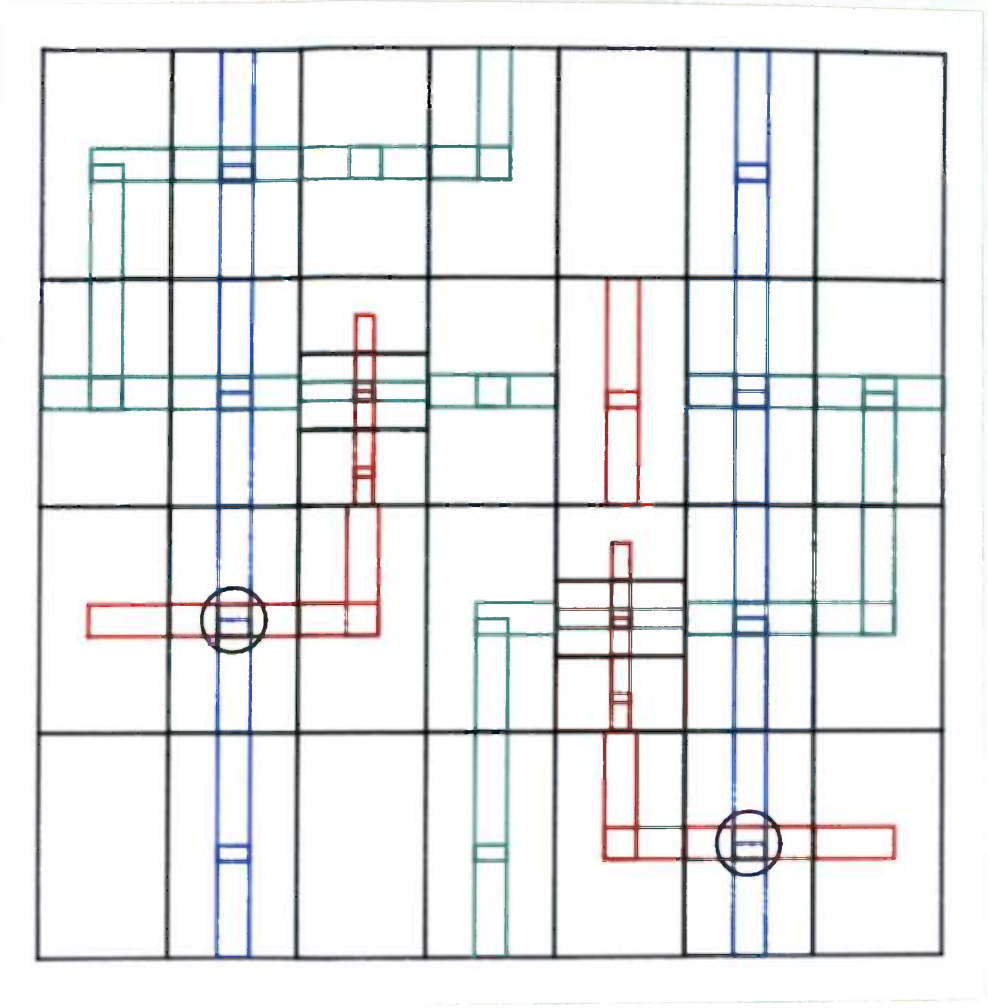


Figure 5–12: An incorrect TallyUnit

The | character indicates horizontal abutment, the ^ character indicates vertical abutment. The first two errors are caused by the lack of ports for the diffusion wires on either side of the new cell, while the bottom two errors are caused by the extra red wires on the top and bottom of the new cell. The error messages are not as clear as they might be – interpret the first message as *horizontal port clash between the row which has 3 tiles abutted to a green wire at orientation 1 and a red wire (with no added orientation) which has no ports.* The major obfuscation here is determining which ports the error applies to, however, this is not a major problem since the system knows where the semantic error occurs in both the textual-un-parsing and the screen image (and it is also obvious

that it was the last sub-tree replacement that introduced the error). The quality of the error messages becomes more important with larger sub-tree replacements, and more sophisticated actions are called for. Although not implemented in the current version of Wright, the obvious thing to do is to highlight the areas on the screen and graphics display which correspond to attribution errors occurring in the attributed parse tree.

To illustrate the iteration and horizontal grouping language features a 4 input `Tally` cell is given in Figure 5–13, and the corresponding picture is given in Figure 5–14. Iterated cells are only shown once, with the remaining area overlayed by an *iteration tile* which indicates the number of repeated cells with an appropriate number of dots. The user can interactively control the amount of detail that is shown in the picture by altering the *depth* of cell instantiation. Figure 5–15 shows a fully instantiated 4 input tally cell.

In this example the cell headers for `routeS`, `route` and `Tally` do not contain port names, the external port appearance is taken to be that of their respective `AbutBlock`. Figure 5–16 shows how a pad placement stage could be specified using Stick-Wright. The text at the start of the screen is the end of the `Cor` (corner cell) definition.

The pad placement example shows how a quite complex arrangement of cells can be specified in a succinct manner. The use of implicit port connection leads to this, while the type checking attributes help prevent un-intended connections. The main reason for adopting abutment over explicit port connection (c.f. Sticks&Stones [13]) is that it leads to shorter descriptions, and is hence easier to write and modify. The explicit connection strategy may prevent some un-intended connections being made which are let through by the Stick-Wright scheme (i.e. legal syntactic connections, but not what the user meant), however, the added complication in the explicit connection description may lead to mistakes in the specification itself. The added responsibility of dealing with geometric transformations for the user in Stick-Wright has not proven to be too burdensome, and is in keeping with the notion that circuit structures should be conceived with the planarity of the design space firmly in mind.

```
Cell TallyUnit (in1:d;
                Xbar:m, out2:d, X:m;
                out1:d;
                X:m, in2:d, Xbar:m ) =
  Abut gc^, gbx, g^, gc^^^, , b, ;
       gt, gbx, <rs^^, pass, r>, g^, g^, gbx,gt^^;
       rs^, rbc, rc^^^, gc^, <rs^^, pass, r>, gbx, gc^^^;
       , b, , g, rc, rbc, rs^^^

Cell routeS (;
             ;
             ;
             ) = Abut , b, , , , b, ;
                      g^, gbx, g^, g^, g^,gbx, g^;
                      , b, , , , b, ;
                      , b, , gs^^, , b,

Cell route (;
            ;
            ;
            ) = Abut , b, , , , b, ;
                     g^, gbx, g^, g^, g^,gbx, g^;
                     , b, , , , b, ;
                     , b, , , , b,                .

Cell Tally (;
            ;
            ;
            ) = Abut (, b, , , , b, )[3], (, b, , gs^^, , b, ) ;
                     route[2], routeS, TallyUnit;
                     route, routeS, TallyUnit[2];
                     routeS, TallyUnit[3];
                     TallyUnit[4];
                     TallyUnit[4]                                .
```

Figure 5–13: 4 input Tally (program)

5.6 Extended Stick-Wright

The implementation of Stick-Wright presented in the last section can always directly derive a picture (or more accurately, the information needed to draw a picture) from any `Cell` definition. When parameterisation is added to the language this is no longer the case; what does a `Tally {n}` look like ? Presently,

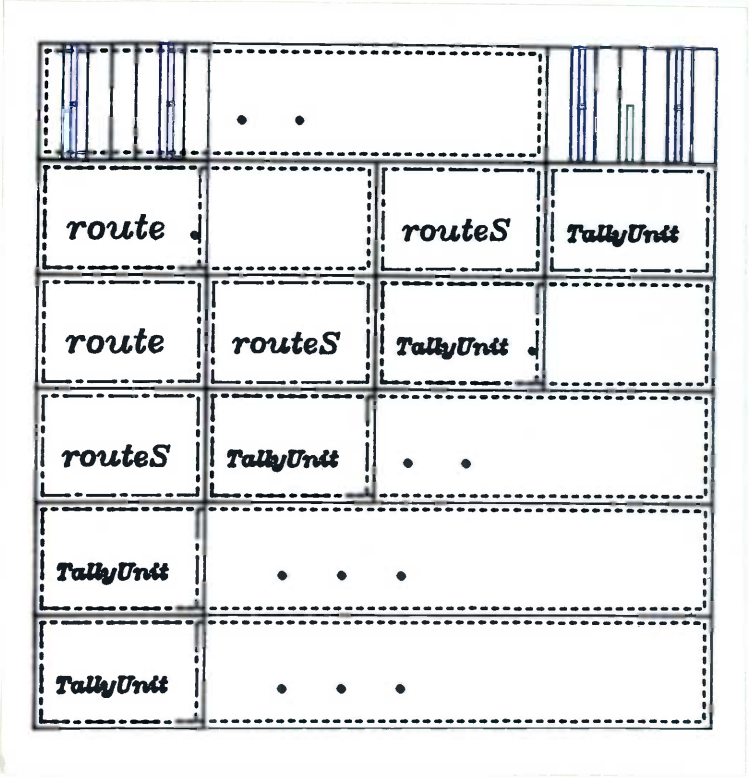


Figure 5–14: 4 input Tally (picture)

most graphics systems address this problem by delaying any drawing until all the parameters have been evaluated, and so a particular instantiation is drawn.

The introduction of variable expressions would also demand changes in the current implementation scheme; values of expressions could not be stored as attributes in the parse tree since cell definitions can be used many times with different sets of values. Variables would have to be represented as address references to a data-stack, and hence expression evaluation would be delayed until the cell was to be instantiated. The attribute evaluation stage would therefore act like a programming language compiler, with the attributed syntax tree corresponding to the generated machine code, and the tree-walking picture generation stage corresponding to the machine interpretation of that code. Parse tree interpretation is a technique which has successfully been used in other systems, e.g. the OCCAM interpreter implemented by Marshall [52] derives its interpretation data-structure directly from the parse tree constructed by an APG [53] parser.

Although full picture instantiation is often desirable, there is an intermediate notational stage available to answer the question *what does* `Tally {n}` *look like*

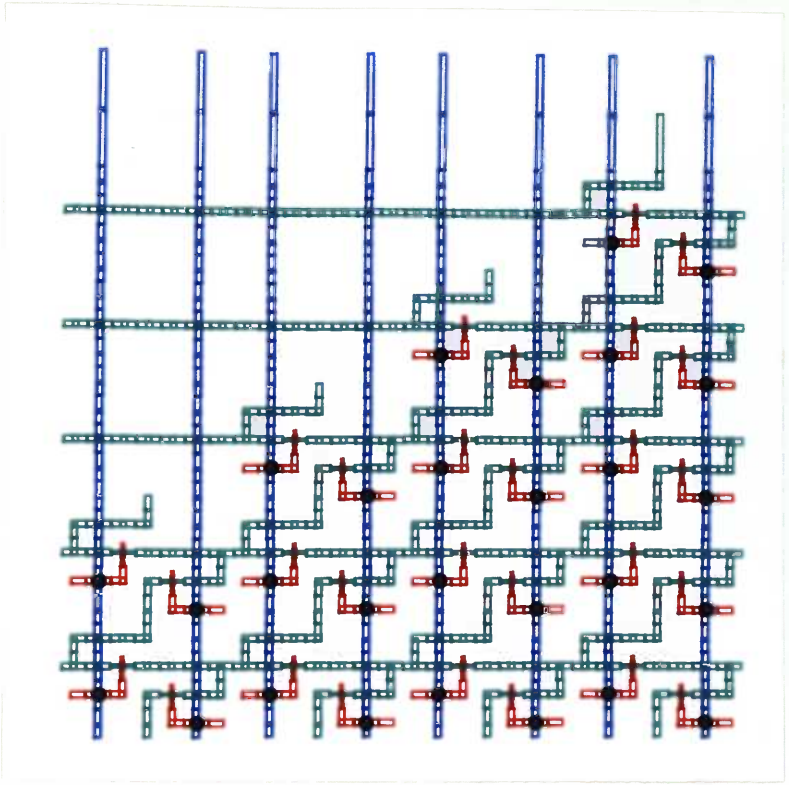
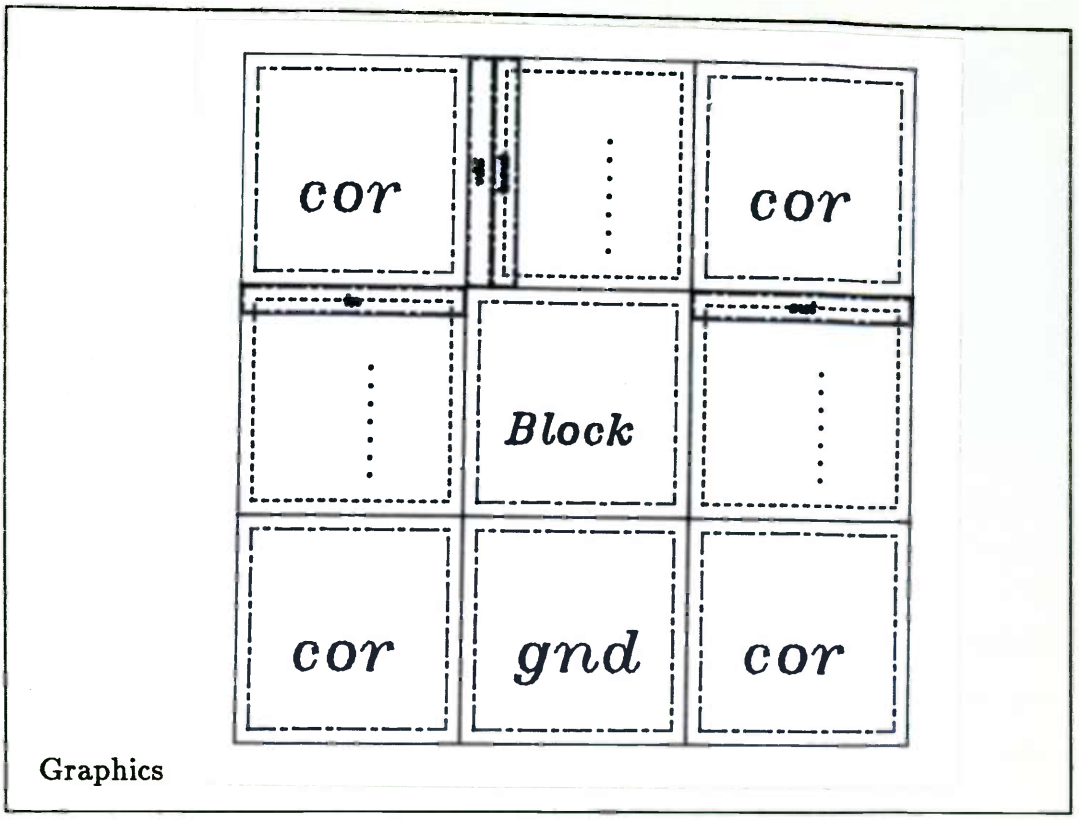


Figure 5–15: 4 input Tally (picture (full instantiation))

? Indeed, this representation was shown in Figure 5–1 on page 99. Quite simply, the `Tally {n}` cell can be viewed as being a box with labelled wires coming out of it with buses being differentiated from single wires by having a thicker line representation. This picture conveys all the information relevant for using the cell in further compositions, but does not show its implementation. Figure 5–17 shows an Extended Stick-Wright implementation of the `Tally cell`, (which uses the cells `TallyUnit`, `routeS` and `route` defined in Figure 5–13. The present implementation of Stick-Wright produces the picture shown in Figure 5–18 for this example. Iteration by a variable value is represented by one instance of the iterated cell overlayed by the iteration box, with two dashes representing a variable range. The conditional statement is expanded with the true case having been assumed, an alternative to that would be to show both conditional cases at a smaller scale.

In the version of Stick-Wright described in the last section, the cell exterior used when instantiating a cell in an `AbutBlock` is provided by the called cell's own `AbutBlock`, not its header. Cell headers are not used in the current




```
——Stic————————————————————————————————————————————————————————————————————
          gnd:m, vdd:m ) = Abut bc^^^, b^, b^;
                                    b, , ;
                                    b, , bc^^^ .

Cell Block (in:d[8];
            vdd:m, (en:p, out:p, in:m)[8];
            out:p[8];
            gnd:m                                  ) = Abut , (bs, (bs, rs, rs)[8]), ;
                                                              <gs^^^[8]>, , <rs ^[8]>;
                                                              , bs^^,                .

Cell Chip (;
           ;
           ;
           ) = Abut cor, (vdd, inout[8]), cor^;
                    <in^^^[8]>, Block, <out^[8]>;
                    cor^^^, gnd ^^, cor^^        .

Chip
————————————————————————————————————————————————————————————————————————————
——Command———————————————————————————————————————————————————————————————————

p<Design   1>>
```

VDU

Figure 5–16: Pad-Placement Example

```
Cell Col {i, n }(;
                 ;
                 ;
                 ) = Abut [i=n -> (bs^^, gs^^, bs^^)|(bs^^, bs^)];
                          <[i<n-1 -> route[n-i-1]|(b, g, b)]>;
                          [i<n -> routeS|(b, g, b)];
                          <TallyUnit[i]>

Cell Tally {n}(;
               ;
               ;
               ) = Abut Coli+1, n+1[i := 1 ..  n].
```

Figure 5–17: Tally {n} (program)

implementation of Stick-Wright, and hence only port constraints generated by the primitive cells are available for use in composition checking (they are the layer names `m`, `d` and `p`. In an implementation of Extended Stick-Wright the cell exterior would be obtainable directly from the cell header. The presence of parameters in any of the port expressions for that cell would require evaluation, but the cell itself would not have to be constructed to provide the exterior profile. This does, however, make the assumption that the cell's header is an accurate statement of what the implementing `AbutBlock` will provide for any set of parameters. With the proposed scheme either the user's specification has to be trusted, or else every cell instantiation has to be fully expanded (this would happen in any case, when the final design is expanded prior to being passed on to further processing stages). An other alternative would be to heavily constrain the use of parameters in the `AbutBlock` implementation, making the port exterior statically derivable (directly from attribution rules).

Ports can be labelled with lists of conditions which all have to be present in both ports during a port connection, e.g. a port `vdd:m:VDD` could connect to a port `vdd1:m:VDD` but not to a port `vdd1:m`, since the latter port does not have the constraint name `VDD`. Static Stick-Wright uses only layer constraint names, as generated by the primitive cells, but the mechanism for multiple constraint

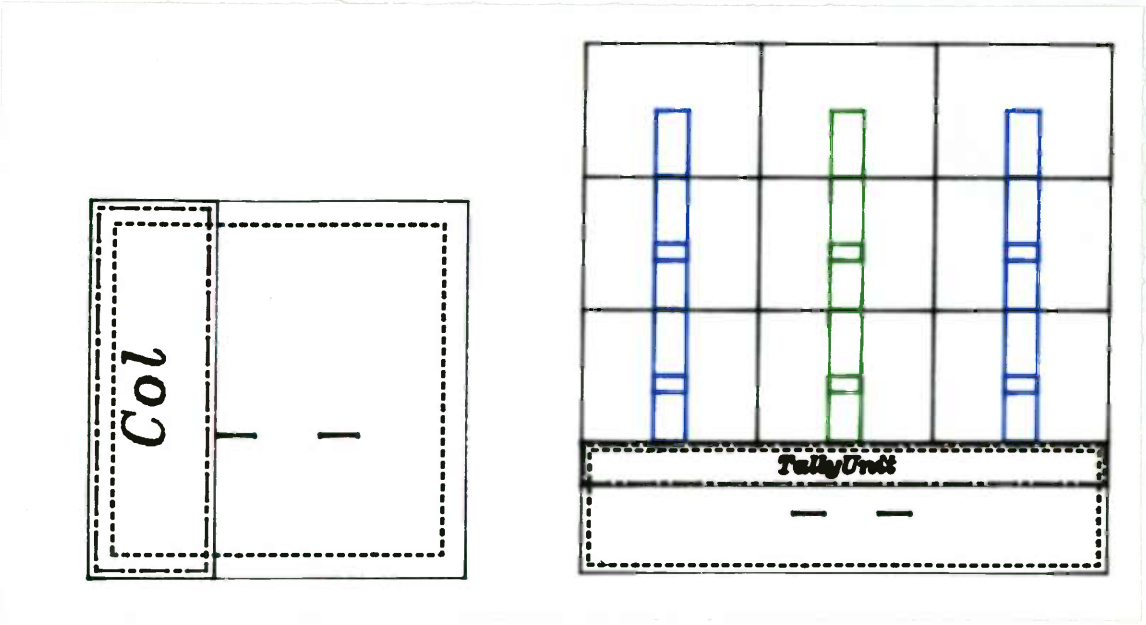


Figure 5–18: Tally {n} and Col {n} (pictures)

lists is in place. In addition the following scheme is proposed for Extended Stick-Wright:

```
Cell or ((in1,in2):m:>out;; out1:m:<out) = ...
```

Here ports are given constraint names which have added directionality. The > operator indicates a port constraint name which can only be connected to a port with the same port constraint name labelled with the < operator. In the above example, the output ports of or cells can be joined to the input ports of or cells, but inputs can not join inputs, nor outputs join with outputs. This is a simple extension, more complex constraint expressions, e.g. due to the addition of a choice operator, could easily be added to the AG.

Chapter 6

Results, Conclusions and Extensions

6.1 The Wright System

The major piece of work undertaken for this thesis was the design and implementation of the Wright generator for language-based graphical editors. The primary motivation behind the development of the Wright system was the desire to provide a formal basis for the development of text/graphic tools. The syntax and semantic analysis stages of this system owe much to work reported by other researchers in programming environments, however, the graphical interface and the use of an attributed-syntax-tree as a graphical-data-structure are novel contributions in this area.

The system was developed from scratch, for the simple reason that no other tools were available at the outset, however, this provided the opportunity for making the integration of text and graphics the primary design consideration. In comparison with other systems, notably the Cornell Synthesizer Generator, the attribute evaluation algorithm and attribute storage strategies are rather inefficient. However, they have proved adequate for the purposes of this thesis and are repairable deficiencies. Wright has succeeded in demonstrating the efficacy of attribute-grammar specification techniques to the text/graphic problem domain, and has developed into a useful tool for the production of working systems. The syntax and lexical analysis components of the Wright system have also successfully been used in a number of other research projects, including an OCCAM compiler and a PLA based silicon compiler.

The use of compiler-compiler systems for generating graphical tools was suggested as long ago as 1967 by Kulsrud [41]. While improvements in hardware since then have been quite dramatic, the development of such compiler-compiler technology has been decidedly less so, and very few systems see much use outside universities. Perhaps the major reason for this is that very few systems undergo the code refinement necessary to meet production quality standards, and that it is generally perceived to be easier to hand-craft one-off systems, rather than invest the effort in a more general approach.

For a compiler-compiler to be a product, its output must also be of production quality. This implies greater attention to issues like efficient storage schemes, and space/time tradeoffs in table compaction. For such a refinement process to be undertaken in a commercial setting, the compiler-compiler must be seen to be in demand. Reasons for wanting formally specified, automatically generated systems were outlined in the first chapter. Briefly restated, they provide ease of implementation (assuming the user is familiar with the formalism being used), security (less bugs) and flexibility. The generated systems themselves, the structure editors and incremental compilers, open up new possibilities in design exploration and verification, as well as spurring on the trend towards more user-friendly, interactive systems. The Wright system allows the relatively fast generation of prototypes as a new system evolves. The generic modules, such as the display and parsing modules, relieve the tool-builder from the necessity of continually having to re-invest effort in implementing front-ends from scratch. As a new tool develops, even partial implementations can be tested using the editing modules.

The limitations of the approach taken in this thesis must also be recognised, formalisms have a habit of breaking down every now and then when applied to real problems. For example, the formalism of regular-expressions does not cope with every lexical item one might like to define, while attribute grammars in their standard form only allow the declaration of local relationships in the parse tree. The response to these kinds of problems is usually to tack on extensions to the formalism which increase the class of candidate problems, e.g. the regular-

expression basis of LEX [20] has added operators for detecting ends of lines and files, and a backtracking operator for re-reading the input. The trouble with extensions is that they can over-complicate the system, and may also remove properties of the formalism that lead to automatic implementation and the application of verification techniques. In the area of lexical analysis and syntactic analysis, the usual *tricks* to increase the utility of the systems do not seriously impact on either the elegance or security of the generated systems; however, more care is needed in the area of semantic functions.

In the Wright system the semantic functions in the attribute grammar are left entirely to the the user as an exercise in normal HLL programming, with the system only responsible for controlling the order of application of the functions, within the context of an interactive editing session. Because the user is allowed the freedom of an unconstrained procedural programming language, there is nothing to prevent the semantic functions being written in a style which makes use of side-effects and shared resources. While this is necessary for improving the storage demands of attributes, it does involve a departure from the strict application of the formalism. This situation can be partially avoided by providing a library of standard semantic functions, e.g. constructors for list structures, and constraining the user to only using predefined operators and structures in the attribute occurrences (this is effectively achieved in the Cornell system by the inclusion of such functions and structures in the grammar specification language itself).

The attribute grammar is a useful descriptive tool, but it is limited in its range of application, e.g. context-sensitive syntax checking (e.g. type-checking) and simple translation and construction processes (e.g. the construction of a graphical data-structure). As such it can play a crucial role in the front-ends of a wide range of design-tools.

6.2 Stick-Wright

The Stick-Wright editor serves two purposes; it is a working demonstration of a Wright generated programming environment, and it also contributes some new ideas to the area of VLSI design tools. The essential feature of the system is that the major data-structure of the editor, the attributed syntax tree, is used to represent both a textual and a pictorial representation of the object being designed (in this case, an IC stick-diagram). This close coupling between text and graphics is crucial to the provision of an interactive design environment in which the user can control the development of a design using a variety of editing techniques.

The first method of graphical interaction demonstrated in this thesis was the graphics text macros developed for Pict-Wright. These macros allow the editor designer to structure graphical actions (e.g. mouse movements, menu selections etc.) by associating them with editor keys (which initiate the graphical command) and also with textual insertions into the program text. This provides a way for developing graphics commands which modify the design using the language interface (and hence guarantee a correct internal representation). The second type of graphical action does not necessarily cause text insertions (although it could initiate them), but provides a way for graphical interactions to cause changes in the status of the current edit, e.g. changing the current tree position, or repainting parts of the tree.

The use of syntactic and semantic constraints to enforce a design methodology has great potential for controlling the complexity of a VLSI design [70]. Stick-Wright demonstrates an effective implementation technique for applying such design strategies.

Graphical notations for dealing with programming language features have been presented; iteration of cells within a cell composition is implemented by the *iteration tile* in Stick-Wright and conventions for dealing with conditions

and parameters, without recourse to full instantiation, are suggested in Extended Stick-Wright.

Stick-Wright, although a small system with a compact specification, has many powerful features and is a significant step towards a VLSI programming environment that fully supports design exploration and verification.

6.3 Extensions

6.3.1 Structure Editing

The generic editor module in the Wright system provides a basic selection from the tree and text editing commands commonly found in such systems, and could easily be extended to include a wider range. The graphical editing commands tend to be more application specific, although common techniques could be shared between systems (e.g. the geometrical transformation attributes of Stick-Wright could be adapted to implement a variety of graphical data-structures).

A major component of the Wright system's text editing module is the pretty-printer which displays the textual version of the attributed syntax tree. At the moment the pretty-printer produces a flattened version of the current position and surrounding text, thus partially loosing the hierarchy implicit in the original tree, although the tree structure is indicated by the highlighting of the current sub-tree. It would be desirable to have more complicated schemes available which were more selective of what they displayed, e.g. reducing detail (e.g. procedure bodies) when displaying text outside the the current area of interest. The attribute grammar presents a direct means for allowing the grammar designer to control such activities, if appropriate printing primitives are provided. By viewing a program text as both a hierarchical and dynamic structure, it should be possible to maintain a view of the design which keeps the user fully briefed on currently salient features, thus speeding up the design process. Just how such views can be provided, and whether the attribute grammar is the best way

to specify their automatic generation, remain an interesting problem. Related work in the area of pretty-printing by Rose and Welsh [74] and also Woodman [96], are folding and indentation algorithms for displaying the text in a fully flattened form. These algorithms are more adaptive than the simple fixed parameter approach taken in Wright and could either be applied to the text structures generated by the attribution stage, or even integrated into the semantic functions themselves.

The use of parsing tables for the auto-completion of tokens and phrases (as described in Chapter 3) is another area where the editing modules of Wright could be extended; the information for performing such activities is freely available to the generic modules. A working auto-completion module was written for ASG (the lexical analyser), but has not yet been adapted for inclusion into the Wright system.

6.3.2 Physical Design

In Stick-Wright the attribute grammar specification is mostly concerned with the decoration of the parse tree for graphical display and the syntactic checking of port compositions. Both these areas are extremely important to any CAD system, and are certainly candidates for further development. The standard symbol-table, expression compilation and pretty-printing attribution techniques can apply to a wide range of input-languages, and present the opportunity for a range of related design tools to share implementation modules. In this and the following sections I consider wider areas of application for the AG specification technique.

Stick-Wright is only one possible approach to a symbolic layout system. Its tiling/abutment strategy could be replaced by an explicit port-matching/wiring approach (as in Sticks&Stones), indeed, a mixture of these techniques was originally considered for Stick-Wright. Further developments to the dedicated editing functions could be made, e.g. program transformations which alter the external appearance of a cell to conform to some new positional context (possibly by the

automatic recursive re-arrangement of cell interiors (as suggested by Cardelli [12]), or by the addition of routing channels. A graphical representation of dynamic language features is suggested in Extended Stick-Wright, but has yet to be tested in a full implementation.

The graphical data-structure supported by the Stick-Wright suggests a foundation for the implementation of many other graphical tools, e.g. the schematic entry system described in section 2.2.5. In schematic entry the physical positioning of the symbols in the schematic is not meant to suggest a specific placement in the generated layout, but it is important that the symbols and the connecting wiring do have a clear and understandable layout in the actual schematic. Although algorithms have been developed which generate pleasing wiring patterns and component placements (and those algorithms will continue to be important), the user can significantly contribute to a good schematic layout by judicious graphical placement with a pointing device. By providing an appropriate set of graphical primitives the system could allow the user to enter an entire design without even being aware of the text version, however, the presence of the textual interface may also encourage the use of programming features like parameterisation and conditional evaluation, perhaps using the graphical notational devices suggested in the previous chapter. The close coupling between graphics and language should also aid in the maintenance of any such system, since the implementation is effectively driven from one single specification, the attribute grammar.

6.3.3 Silicon Compilation

Perhaps the greatest demand for the automatic generation of language processors (at least in the area of VLSI) will come in the emerging field of silicon compilation. The reasons for this are simply that it is unlikely that any particular hardware description language (HDL) will dominate for some time to come, and that many attempts at finding the best way to specify hardware systems will have to be made. Not only are the specifications of such systems a mov-

ing target, the underlying technologies are also continually evolving. The quick production of prototype systems offered by systems such as Wright may be an answer to these problems.

Many current attempts at HDLs (e.g. VHDL [47]) bear strong resemblances to either Pascal or ADA, with added keywords like `Signal` and `After`. If this continues to be a trend then the attribute grammar specification work on such procedural languages [89] will prove useful in generating appropriate programming environments for such systems. Other types of programming language, including applicative and object-oriented, can also benefit from the AG approach, and may also be explored as candidates for implementing HDLs.

6.3.4 Verification and Simulation

Stick-Wright demonstrates a syntactic method for restricting the domain of allowable circuit structures. This approach can be extended to deal with further physical, structural and behavioural properties, e.g. Milne's calculus for circuit descriptions, Circal [58], could be implemented using an AG, and program transformations and editing commands could be developed for manipulating the Circal expressions. A major problem in the formal language approach to VLSI verification has been the size of the expressions which represent relatively simple structures, and the difficulty in performing equivalence proofs on such expressions. Proof editing has been the subject of investigation for a number of AG projects, and is an active area of research. Machine assistance will be of paramount importance in the completion of any large scale verification.

An appealing design system which would make full use of the graphical facilities provided by the Wright system is an interactive simulator using a hardware description language and circuit model [60] to animate a design specification on a graphics screen. The structure editor and graphics commands could be used for editing not only the circuit description, but also the driving simulation script. The techniques developed in this thesis provide a means for building just such an interactive design environment.

Bibliography

[1] A.V. Aho and J.D. Ullman. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1979.

[2] F. Anceau. LSI–processor architecture. 1984. Presented at the NATO Advanced Study Institute on Microarchitecture of VLSI Computers, Urbino, Italy.

[3] F. Anceau. Statistical properties and layout strategies for NMOS and CMOS layout. 1984. Presented at the NATO Advanced Study Institute on Microarchitecture of VLSI Computers, Urbino, Italy.

[4] H.G. Barrow. Proving the correctness of digital hardware designs. *VLSI Design*, 5(7):64–77, July 1984.

[5] N. Bergmann. *Idiomatic Integrated Circuit Design.* PhD thesis, University of Edinburgh, August 1984.

[6] G.H. Birtwistle, O.J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA* `begin`. Auerbach Publishers Inc., Philadelphia, 1973.

[7] G. Brebner and D. Buchanan. On compiling structural descriptions to floorplans. In *International Conference on Computer Aided Design*, pages 6–7, 1983.

[8] D. Buchanan. personal communication, 1985.

[9] I. Buchanan. *Modelling and Verification in Structured Integrated Circuit Design.* PhD thesis, University of Edinburgh, 1980.

[10] M. Burich. SDL compiler compiler – the design of module generators. July 1986. Presented at the NATO Advanced Study Institute on Logic Synthesis and Silicon Compilation for VLSI Design.

[11] R.H. Campbell and P.A. Kirslis. The SAGA project: a system for software development. *ACM SIGPLAN Notices*, 19(5):73–80, May 1984.

[12] L. Cardelli. *An Algebraic Approach to Hardware Description and Verification.* PhD thesis, University of Edinburgh, 1982.

[13] L. Cardelli. *Sticks and Stones: An Applicative VLSI Design Language.* Technical Report CSR–85–81, University of Edinburgh Department of Computer Science, June 1981.

[14] L. Cardelli. *Two–Dimensional Syntax for Functional Languages.* Technical Report CSR–115–82, University of Edinburgh Department of Computer Science, May 1982.

[15] J. Cherry, H. Shrobe, N. Mayle, C. Baker, H. Minsky, K. Reti, and N. Weste. Ns: an integrated design system. In E. Horbst, editor, *VLSI 85*, pages 325–334, August 1985.

[16] P.B. Denyer, D.A. Renshaw, and N. Bergmann. A silicon compiler for VLSI signal processors. In *Digest of Technical Papers*, pages 215–218, ESSCIRC, 1982.

[17] V. Donzeau–Gouge, G. Huet, G. Kaha, and B. Lang. *Programming Environments based on structured editors: the Mentor experience.* Technical Report, INRIA, France, May 1980.

[18] A.E. Dunlop. Slim — the translation of symbolic layouts into mask data. In *ACM IEEE 17th Design Automation Conference*, pages 595–602, 1980.

[19] *EDIF: Electronic Design Interchange Format, Version 1.0.0.* Electronic Design Interchange Format Steering Committee, 1985.

[20] Lesk E.M. and Schmidt E. *Lex: A Lexical Analyser Generator.* Bell Laboratories, 1978. UNIX Programmer's Manual.

[21] C.N. Fischer and et al. The POE language–based editor project. *ACM SIGPLAN Notices*, 19(5):21–29, May 1984.

[22] J.D. Foley and A. Van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, Reading, Mass., 1982.

[23] J.G. Gay, R. Richter, and B.J. Berne. Component placement in VLSI circuits using a constant pressure monte carlo method. *Integration, the VLSI Journal*, 3(4):271–282, 1985.

[24] A. Goldberg and D. Robson. *SMALLTALK-80.* Addison-Wesley, Reading, Mass., 1983.

[25] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1979.

[26] J.P. Gray, I. Buchanan, and P.S. Robertson. Designing gate arrays using a silicon compiler. In *ACM IEEE 19th Design Automation Conference*, pages 377–383, 1982.

[27] Brown H., Tong C., and Foyster G. Palladio: an exploratory environment for circuit design. *Computer*, 16(12):41–56, December 1983.

[28] W.R. Heller, Sorkin G., and K. Maling. The planar package planner for system designers. In *ACM IEEE 19th Design Automation Conference*, pages 253–260, 1982.

[29] J.G. Hughes. *The Edwin User's Guide (Fourth Edition).* Technical Report CSR–74–81, University of Edinburgh Department of Computer Science, August 1981.

[30] J.G. Hughes. The ILAP library. VLSI Design Tools Volume 1, University of Edinburgh Department of Computer Science.

[31] F. Jalili. A general incremental evaluator for attribute grammars. Moore School of Electrical Engineering, Philadelphia, March 1983.

[32] F. Jalili. A general linear-time evaluator for attribute grammars. *ACM SIGPLAN Notices*, 18(9):35–44, 1983.

[33] G.F. Johnson and C.N. Fischer. Non-syntactic attribute flow in language based editors. In *9nth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 185–195, 1982.

[34] S.C. Johnson. *Yacc: Yet Another Compiler-Compiler.* Bell Laboratories, 1978. UNIX Programmer's Manual.

[35] S.G. Johnston. *Graphical Display of a Concurrent Device Simulation using CIRCAL.* Technical Report CSR–204–86, University of Edinburgh, Department of Computer Science, August 1986.

[36] L.G. Jones and J. Simon. Hierarchical VLSI design sytems based on attribute grammars. In *14th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1986.

[37] R.K. Jullig and F. DeRemer. Regular right–part attribute grammars. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 171–178, June 1984. SIGPLAN Notices Vol. 19, No. 6.

[38] D.E. Knuth. On the translation of languages form left to right. *Information and Control*, 8(6):607–639, 1965.

[39] D.E. Knuth. Semantics of context–free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. correction in vol.5,1, p.95–96, 1971.

[40] P.W. Kollaritsch and N.H.E. Weste. A rule–based symbolic layout expert. *VLSI Design*, 5(8):34–42, August 1984.

[41] H. E. Kulsrud. A general purpose graphic language. *Communications of the ACM*, 11(4):247–254, April 1968.

[42] F. Lakin. Computing with text-graphic forms. In *LISP Conference, Stanford*, pages 100–106, 1980.

[43] R.P. Larsen. Computer–aided preliminary layout design of customized MOS arrays. *IEEE Transactions on Computers*, C–20(5):512–523, May 1971.

[44] *Chipsmith, a random logic compiler for gate arrays, optimised arrays and standard cells.* Lattice Logic Ltd., 9 Wemyss Place, Edinburgh EH3 6DH, 1985.

[45] H.S. Law and G. Wood. A mixed–media approach to module generator design. July 1986. Presented at the NATO Advanced Study Institute on Logic Synthesis and Silicon Compilation for VLSI Design.

[46] T. Lenguaer and K. Mehlhorn. The HILL system: a design environment for the hierarchical specification, compaction, and simulation of integrated circuit layouts. In P. Penfield, Jnr., editor, *Conference On Advanced Research in VLSI*, pages 139–149, Massachusetts Institute of Technology, 1984.

[47] R. Lipsett, E. Marschner, and M. Shahdad. VHDL – the language. *IEEE Design & Test*, 3(2):28–41, April 1986.

[48] R.J. Lipton, J. Valdes, G. Vijayan, S.C. North, and R. Sedgewick. VLSI layout as programming. *ACM Transactions on Programming Languages and Systems*, 5(3):405–421, July 1983.

[49] B. Locanthi. A simula package for IC layout. Caltech Display File #1862, 1978.

[50] B. Lorho, editor. *Methods and Tools for Compiler Construction.* Cambridge University Press, 1984.

[51] K. Maling, S. Mueller, and W.R. Heller. On finding most optimal rectangular package plans. In *ACM IEEE 19th Design Automation Conference*, pages 663–670, 1982.

[52] R.M. Marshall. Automatic generation of controller systems from control software. To appear at the International Conference on Computer Aided Design, November 1986.

[53] G.A. McCaskill. *APG: An Automatic Parser Generator.* University of Edinburgh Department of Computer Science, 1985.

[54] G.A. McCaskill. *Interactive ILAP.* Technical Report CSR–147–83, University of Edinburgh, Department of Computer Science, October 1983.

[55] C.A. Mead. VLSI and technological innovations. In J.P. Gray, editor, *VLSI 81*, pages 3–11, August 1981.

[56] C.A. Mead and L.A. Conway. *Introduction to VLSI Systems.* Addison-Wesley, Reading, Mass., 1980.

[57] R. Medina–Mora and P.H. Feiler. An incremental programming environment. *IEEE Transactions on Software Engineering*, SE–7(5):472–482, September 1981.

[58] G.J. Milne. Circal: a calculus for circuit description. *Integration, the VLSI Journal*, 1(2,3):121–160, 1983.

[59] G.J. Milne. *The Correctness of a Simple Silicon Compiler.* Internal Report CSR–127–83, University of Edinburgh, Department of Computer Science, January 1983.

[60] G.J. Milne. A model for hardware description and verification. In *ACM IEEE 21st Design Automation Conference*, pages 251–257, IEEE, June 1984.

[61] W.M. Newman and R.F. Sproull. *Principles of Interactive Computer Graphics.* McGraw–Hill, New York, 1973.

[62] A.R. Newton. Symbolic layout and procedural design. July 1986. Presented at the NATO Advanced Study Institute on Logic Synthesis and Silicon Compilation for VLSI Design.

[63] T. Ng and S.L. Johnsson. Generation of layouts from mos circuit schematics: a graph theoretic approach. In *ACM IEEE 22nd Design Automation Conference*, pages 39–45, IEEE, June 1985.

[64] I.M. Nixon. Chip churn: a PLA based silicon compiler. To appear at the International Conference on Custom and Semi–Custom ICs, November 1986.

[65] D.C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):466–483, October 1980.

[66] R.H.J.M. Otten. Automatic floorplan design. In *ACM IEEE 19th Design Automation Conference*, pages 261–267, 1982.

[67] J.K. Ousterhout, G.T. Hamachi, R.N. Mayo, W.S. Scott, and G.S. Taylor. Magic: a VLSI layout system. In *ACM IEEE 21st Design Automation Conference*, pages 152–159, IEEE, June 1984.

[68] C. Piguet, E. Dijkstra, and G. Berweiler. Automatic generation of CMOS layout cells from a hardware description language. In K. Waldschmidt and B. Myhrhaug, editors, *Euromicro 85*, pages 477–486, September 1985.

[69] M. Rem. The VLSI challenge: complexity bridling. In J.P. Gray, editor, *VLSI 81*, pages 65–73, August 1981.

[70] M. Rem and C.A. Mead. A notation for designing restoring logic circuitry in CMOS. In Charles L. Seitz, editor, *Proceedings of the Second Caltech Conference on Very Large Scale Integration*, pages 399–411, January 1981.

[71] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[72] M.C. Revett and P. Ivey. Astra — a CAD system to support a structured approach to IC design. In F. Anceau and A.J. Aas, editors, *VLSI 83*, pages 413–422, August 1983.

[73] P.S. Robertson. *The IMP77 Language.* Technical Report CSR–19–77, University of Edinburgh Department of Computer Science, November 1980.

[74] G.A. Rose and J. Welsh. Formatted programming languages. *Software - Practice and Experience*, 11(7):651–669, July 1981.

[75] J.B. Rosenberg. Chip assembly techniques for custom IC design in a symbolic virtual-grid environment. In P. Penfield, Jnr., editor, *Conference On Advanced Research in VLSI*, pages 213–225, Massachusetts Institute of Technology, 1984.

[76] A. Sangiovanni–Vincentelli. Placement and routing in a synthesis environment. July 1986. Presented at the NATO Advanced Study Institute on Logic Synthesis and Silicon Compilation for VLSI Design.

[77] S. Sastry and S. Klein. Plates: a metric free VLSI layout language. In P. Penfield, Jnr., editor, *Conference On Advanced Research in VLSI*, pages 165–174, Massachusetts Institute of Technology, 1982.

[78] M. Schlag, Y.Z. Liao, and C.K. Wong. An algorithm for optimal two-dimensional compaction of VLSI layout. In *International Conference on Computer Aided Design*, pages 88–89, 1983.

[79] R. Schmid and U.G. Baitinger. *The Role of Floor Plan Tools in the VLSI Design Process.* Technical Report, Institut Fur Technik der Informationsverarbeitung, Universitat Karlsruhe, 1984.

[80] J. Schoellkopf. Lubrick: a silicon assembler and its application to data-path design for fisc. In F. Anceau and A.J. Aas, editors, *VLSI 83*, pages 435–455, August 1983.

[81] M.J. Siskind, Southard J.R., and Crouch K.W. Generating custom high performance VLSI designs from algorithmic descriptions. In P. Penfield, Jnr., editor, *Conference On Advanced Research in VLSI*, pages 28–39, Massachusetts Institute of Technology, 1982.

[82] J.R. Southard. Macpitts: an approach to silicon compilation. *Computer*, 6–12:74–82, December 1983.

[83] S. Taylor. Symbolic layout. *VLSI Design*, 5(4):34–42, March 1984.

[84] Whitney T.E. and Mead C. An integer based hierarchical representation for VLSI. In C.E. Leiserson, editor, *Conference On Advanced Research in VLSI*, pages 241–257, Massachusetts Institute of Technology, 1986.

[85] T. Teitelbaum, T. Reps, and S. Horowitz. The Cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981.

[86] R.D. Tennent. Language design methods based on semantic principles. *Acta Informatica*, 8:97–112, 1977.

[87] N. Traub. *A Lisp Based Circal Environment.* Internal Report CSR–152–83, University of Edinburgh, Department of Computer Science, November 1983.

[88] S. Trimberger. Combining graphics and a layout language in a single interactive system. In *ACM IEEE 18th Design Automation Conference*, pages 234–239, 1981.

[89] J. Uhl, S. Drossopoulo, G. Persch, G. Goos, D. Dausmann, G. Winterstein, and W. Kirchgässner. *An Attribute Grammar for the Semantic Analysis of ADA.* Volume 139 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1982.

[90] W.M. vanCleemput. Hierarchical design for VLSI: problems and advantages. In Charles L. Seitz, editor, *Proceedings of the First Caltech Conference on Very Large Scale Integration*, pages 259–274, January 1979.

[91] W.M. Waite and G. Goos. *Compiler Construction. Texts and Monographs in Computer Science*, Springer-Verlag, New York, 1984.

[92] N.H.E. Weste and B. Ackland. A pragmatic approach to topological symbolic IC design. In J.P. Gray, editor, *VLSI 81*, pages 117–129, August 1981.

[93] T.E. Whitney. *Hierarchical Composition of VLSI Circuits.* PhD thesis, California Institute of Technology, 1985.

[94] T.E. Whitney. A hierarchical design–rule checking algorithm. *Lambda*, 2(1):40–43, First Quarter 1981.

[95] J.D. Williams. A graphical compiler for high level lsi design. In *AFIPS Conference Proceedings*, pages 289–295, 1978.

[96] M. Woodman. Formatted syntaxes and modula-2. *Software – Practice and Experience*, 16(7):605–626, July 1986.

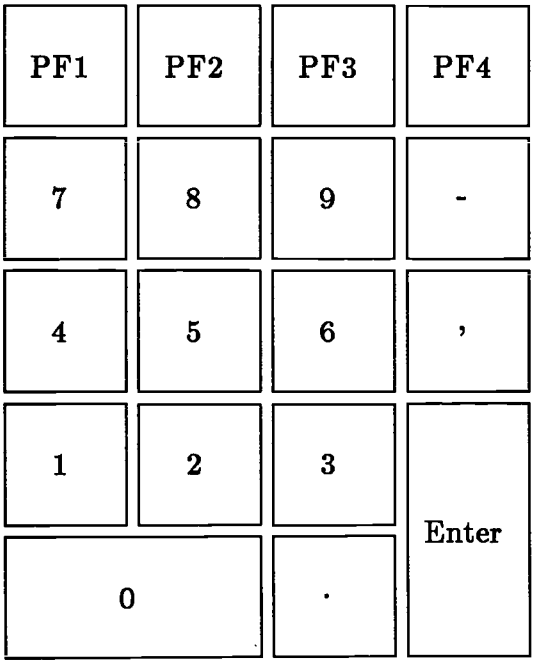
Appendix A

Wright Reference Manual

A.1 Editor Commands

A.1.1 Introduction

This section contains a brief introduction to the editing commands available in the two editors presented in this thesis. The editor commands are assigned to the keypad and cursor keys of the VDU's keyboard. Here is the keypad layout for the VT100 terminal:



In the explanation of the editor commands the keyboard key-name is given before the editor command name. In addition to the keypad names above, most keyboards also have cursor-keys and function keys (F1 ... F13).

A.1.2 Generic–Commands

These commands are found in all Wright generated editors:

cursor-UP — `MoveToFather` move to the father of the current tree node (unless at top of tree).

cursor-DOWN — `MoveToSon` move to first son of current tree node.

cursor-RIGHT — `MoveToRight` move to parent and then down to next son. If the current node is a terminal node and there is no next sibling then traverse the tree up and then down to the next terminal. If the current node is not a terminal and there is not a rightmost sibling, then wrap round to the first sibling.

cursor-LEFT — `MoveToLeft` same as above, but in the other direction.

0 — `MoveToLeftSibling` Same as `MovetoLeft`, but always wrap round at leftmost position to the right-most position.

. — `MoveToRightSibling` Same as `MovetoRight`, but always wrap round at right-most position to the left-most position.

1 — `NextTerminal` Descend down the tree until a terminal is reached.

4 — `DeleteSubTree` Replace the current node with the first child which matches the current production.

6 — `Diagnostics` Print out information on the current node (e.g. attribute values).

7 — `InsertSubTree` Push down current node into parse stack and invoke parser (this command enables list structures to grow).

8 — `ReplaceSubTree` Delete current node and invoke parser to read in replacement.

, — `DoPDFaction` This command calls the user supplied routine which should save the current text and picture to files.

Enter — `ToggleGraphics` This command switches between the graphics window and the text window view on the graphics monitor.

Home — `TopOfTree` Move to root of tree, and also re-evaluate the whole tree.

F3 — `APGDebug` Switch on/off diagnostics for the parser.

F4 — `LexDebug` Switch on/off diagnostics for the lexical analyser.

When the parser is invoked the `Command` window prompts for user input. While in program-entry mode the user can invoke any lexical macros which have been defined in the procedure `do user macro`. Macros can be bound to any keypad key.

The window-manager for the system can be invoked using `Ctrl-W`. Commands for the window manger are:

Cursor Keys move the current window/device about.

Home-Key selects current option (which is displayed in window).

5 execute shell command.

3 dump current window to file.

A.1.3 Pict-Wright

Pict-Wright has no special editing commands, but defines the following lexical insertion macros which can be invoked during program-entry mode.

Enter insert mouse provided `x, y` coordinate pair.

0 insert mouse provided x coordinate.

[.] insert mouse provided y coordinate.

[-] insert a statement `line{x1, y1}(x2, y2)` provided by two mouse positions.

A.1.4 Stick-Wright

Stick-Wright has no lexical insertion macro, but defines the editor commands:

[- — MoveToTile] select a tile in the current graphics image with the mouse. The command then moves the text-cursor to the matching position.

[2 — MoveToCall] move to the `Ident` phrase of the `Design` phrase. This node defines which picture is displayed on the graphics monitor.

[3 — SetDepth] select the depth to which cell calls should be printed (default=1).

[Shifted 3 — SetBounding] Switch bounding boxes on/off.

[9 — Refresh] re-evaluate and re-draw graphics display.

A.2 Wright Input Language

A.2.1 Introduction

This section contains the ASG (scanner generator) and APG (parser generator) specifications of the Wright Input Language, as used for the description of Pict–Wright and Stic–Wright in Chapters four and five.

A.2.2 Lexical Definition

```
Lexical_definition WRT is

  Ranges
```

```
  @L is 'a' .. 'z' + 'A' .. 'Z';
  @N is '0' .. '9';
  @B is 0 .. 32;
  @C is 0 .. 127 - '}';
  @D is 0 .. 127 - ']';
  @A is 0 .. 127 - '>' - @L - '$'
end of ranges

macros
  #case is $$;
  #p is $($)*;
  #o is ($|)
end of macros

expressions

  #case;

  _grammar -> \grammar;
  _is      -> \is;
  _lex     -> \lexicals;
  _code    -> \code;
  _synth   -> \synthesised;
  _inher   -> \inherited;
  _prod    -> \productions;
  _prior   -> \priorities;
  _assoc   -> \associativities;
  _end     -> \end;
  _of      -> \of;
  _right   -> \right;
  _left    -> \left;
  _start   -> \start;

  #;

  _arrow   -> \->;
  _comma   -> \,;
  _semi    -> \;;
  _bar     -> \|;
  _plus    -> \+;
  _star    -> \*;
  _opt     -> \?;
  _lb      -> \(;
  _rb      -> \);
  _user    -> \[@D*\];

  _att     -> \<\ * #p[ @L\ * ] \$ @N ! @N* \ * \=
              #p[
                 @A* #o[ #p[@L\ *] #o[\$ @N ! @N* ]]
               ] \> ;

  _form    -> @B@B*;
```

```
    _comm     -> \{@C*\};
    _name     -> (@L|_)(@L|@N|_)*

  end of expressions

end of lexical_definition
```

A.2.3 Grammar

```
Grammar Wright is

  Code [{}]

  Lexicals _grammar, _is, _lex, _code, _synth, _inher, _prod, _prior,
           _assoc, _end, _of, _right, _left, _name, _start, _arrow,
           _comma, _semi, _bar, _plus, _star, _opt, _lb,
           _rb, _att, _user;

  Productions

    G          -> _grammar _name _is
                    iCode
                    Lexs
                    Synth
                    Inher
                    Prods
                    Priors
                    Ass
                  _end _of _grammar;

    iCode      -> _code imp_code | _error ;

    Lexs       -> _lex Name_list _semi ;

    Synth      -> _synth Def_list _semi |;

    Inher      -> _inher Def_list _semi |;

    Def_list   -> Def_list _comma Def | Def ;

    Def        -> _name _lb Name_list _rb ;

    Name_list  -> Name_list _comma Name_code
                  |
                  Name_code;

    Name_code ->  _name imp_code;

    Prods      -> _prod
                    Prod_list
                  _end _of _prod;
```

```
    Prod_list -> Prod_list Prod | Prod ;

    Prod      -> StartI NT _arrow RHS_list _semi |
                 _error ;

    StartI    -> _start |;

    NT        -> _name ;

    RHS_list  -> RHS_list _bar Alt |
                 Alt;

    Alt       -> RHS imp_code;

    RHS       -> element RHS | ;

    element   -> _name |
                 LB RHS_list RB op;

    LB        -> _lb ;
    RB        -> _rb ;

    op        -> _plus |
                 _star |
                 _opt  |
                       ;


    imp_code  -> imp_code pCode  | ;

    pCode      -> _user | _att ;

    Priors    -> _prior PP _semi | ;
    PP        -> PP PPO | PPO;
    PPO       -> LB Name_list RB | Name_list;

    Ass       -> _assoc Ass_list _semi | ;

    Ass_list  -> Ass_list _comma Ass_spec | Ass_spec ;

    Ass_spec  -> _name _is _left  |
                 _name _is _right ;

  end of productions

end of grammar
```

Appendix B

Attribute Grammar for Pict-Wright

B.1 The Auxiliary Definition File "Pict.src"

This section contains an abridged extract from the auxiliary definition file (which is too long to include in its entirety), and is provided to give an impression as to the implementation of Pict-Wright's semantic functions.

```
!----------------!
! Lexical Actions

%routine lex integer
  ! takes current token character string and evaluates an
  ! integer value from it.  This value is stored in the parse tree
%end

!-----------------------------------------------!
! Lexical Insertion routines

%external %routine %spec REQUEST %alias "EDWIN___F_REQ" -
                         (%integer %name but, x, y)

! finds the current display device cursor position

%routine do user macro (%integer i)
  !
  ! these are the user-supplied graphical interaction routines
  !
  %integer but,x,y,x2,y2

  request(but,x,y) {find position on screen}

  %if i = Keypad Enter %start                  {coord pair}
    give to scanner(coordinate pair(x, y))
  %else %if i = Keypad 0                       {x coord}
    give to scanner(single coordinate(x))
```

```
  %else %if i = Keypad Dot                    {y coord}
    give to scanner(single coordinate(y))
  %else %if i = Keypad Minus
    request(but,x2,y2)
    !                    draw a line from (x,y) to (x2,y2)
    !
    give to scanner(line statement(x,y,x2,y2))
  %finish
%end

%routine do user editor action (%integer i)
  ! none for Pict-Wright
%end


!-----------------------------------------------!
! Pretty Printing Attribution routines
!
%constant %integer default = 30  {maximum size in X-direction}


%record %format text box fm (%short x, y, last x,
                             %byte folds, extra, auto, indent)


%integer %function c0 (%integer size, extra, auto, indent)
  ! make new text box
%end

%integer %function c (%integer box1, size, extra, auto, indent)
  ! make new text box
%end

%integer %function c2 (%integer box1, box2, size, extra, auto, indent)
  ! make new text box
%end

!-----------------------
! Arithmetic Operations

%constant %integer max int = 1000000
                             {values > max int => string address}

%integer %function do negate (%integer a)
  attribution error("type error, expected integer") %if a > max int
  %result = -a
%end

%integer %function do times (%integer a, b)
  attribution error("type error, expected integer") %if a > max int
  attribution error("type error, expected integer") %if b > max int
  %result = a*b
%end
```

```
%integer %function do divide (%integer a, b)
  attribution error("type error, expected integer") %if a > max int
  attribution error("type error, expected integer") %if b > max int
  %result = a // b
%end


%integer %function do plus (%integer a, b)
  %if a > max int %start
    %if b <  max int %start
      attribution error("type error, adding string and integer")
    %finish
    %result = concatenate(a, b)
  %else
    %if b >  max int %start
      attribution error("type error, adding string and integer") -
    %finish
    %result = a + b
  %finish
%end

%integer %function do minus (%integer a, b)
  attribution error("type error, expected integer") %if a > max int
  attribution error("type error, expected integer") %if b > max int
  %result = a - b
%end

%integer %function do int
  %result = integer at first son of current production
%end

%integer %function do string
  %result = address of string at first son of current production
%end

!-----------------------------------------!
! Environment Handling  (ident -> type,value)

%record %format env fm (%integer val,
                        %string(*)%name id,
                        %record(env fm)%name split, next)

%integer %function defadd (%integer a, b)
  %result = add definition lists a and b
%end

%integer %function do binding (%integer a)
  %result = new binding of current _id to value a
%end

%integer %function envadd (%integer a, b)
  %result = addition of environments a and b
```

```
%end

%integer %function def and bind (%integer def)
  %result = create symbol table entry for current _id and
            link with def
%end

%integer %function do bind
  %result = create symbol table entry for current _id
%end

%integer %function initial environment
  %result = the null symbol table
%end

%integer %function do name ref (%integer env)
  %if can find current _id in symbol table env %start
    %result = its value
  %else
    attribution error("identifier _id not declared")
  %finish
%end

%integer %function do length (%integer s)
  ! discover length of string drawing (at current scaling)
%end

!-----------------------------!
! Graphic Data Structure

%record %format pos fm (%integer x, y)

%record %format arg fm (%integer arg, %record(arg fm)%name next)

%integer %function copy origin (%integer i)
  %result =  current value of procedure call origin
%end

%integer %function new origin
  %result = zero origin
%end

%integer %function do colour (%integer arg1, origin)
  ! set colour to arg1
  %result = origin
%end

%integer %function do colour (%integer arg1, origin)
  ! set colour to arg1
  %result = origin
%end
```

```
%integer %function do size (%integer arg1, origin)
  ! set font size to arg1
  %result = origin
%end

%integer %function do font (%integer arg1, origin)
  ! set font to arg1
  %result = origin
%end

%integer %function do move (%integer arg1, arg2, origin)
  ! change current position
  !
  !
  %if arg1 = 0 %start
    new position = origin + arg2    {where + is pairwise}
                                    {addition of coords}
  %else
    new position = origin from parent Command List +
                   arg1 + arg2
  %finish
  %result = new position
%end

%integer %function do line (%integer arg1, arg2, origin)
  ! similar to move, but also draws line
%end

%integer %function do text (%integer arg1, arg2, origin)
  ! similar to move, but also draws text.
  ! arg2 is a string, the length of which is used as a
  ! x-translation, thus leaving the cursor at the end
%end


%integer %function do call (%integer a, arg1, arg2, origin)

  %if id not found in symbol table(a) %start
     attribution error("procedure ".id." not declared")
  %else

    insert parameters into symbol table(proc, arg2)
    ! gives attribution error if too many parameters, or too few.

    move origin(proc, arg1)
    zero tree(proc found for id)

    evaluate tree(proc, Command List Pos) {call evaluator}

    %result = final position after call

  %finish
  %result = origin
```

```
%end

!----------------------------------------!
! Argument List

%integer %function add vals (%integer a, %integer  b)
  ! build arg list
%end

%integer %function add val (%integer a)
  ! initial arg list
%end

!________________!

%integer %function command ref (%integer definitions)
  %result = address of CommandList
%end

%routine initialise user globals
  ! called by parser to initialise user variables
%end


%routine do user eval actions
  ! chance for user to supply graphics commands
  ! prior to tree evaluation (e.g. erasing the display)
%end

%routine do user post eval actions
  ! chance for user to supply commands
  ! after evaluation (e.g. printing of attribute values)
%end

%routine do user move actions
  ! chance for user to supply commands
  ! after a cursor move (like calling the pretty--printer
%end

%routine do user pdf action
  ! user supplied routine for preparing hard copy
  ! of the current graphics display
%end

!--------------------!
! External Interface
!

%external %routine Pict parse (%string(255) file)
  !
  !  application program calls this
  !
  parse(file.".pct")  {the generic parser}
```

```
%end

%end %of %file
```

B.2 The Grammar

```
Grammar Pict is

  Code [%include "pict.src"]

  Lexicals _id,_clb,_crb,_slb,_srb,_comma,_ass,
           _define,
           _line,
           _colour,
           _size,
           _font,
           _text,
           _move,
           _lb,_rb,_minus,_times,
           _length,
           _div,_plus,_int [lex integer],_string;

  Synthesised
    Design  (box, def, pos),
    CommandList (box, def, pos),
    Command (box, def, pos),
    Arg1 (box,vals),
    Arg2 (box,vals),
    ArgL (box,def),
    NL   (box,def),
    List (box,vals),
    Defn (box,val),
    Item (box,val);


  Inherited
    CommandList (env,origin),
    Command (env,origin),
    Arg1 (env),
    Arg2 (env),
    NL   (env),
    List (env),
    Defn (env,origin),
    Item (env);

  Productions

    Design -> CommandList
                <box$0 = c(box$1, default, 0, 0, 0)>
```

```
                <pos$0 = pos$1 >
                <def$0 = def$1 >
                <env$1 = initial environment >
                <origin$1 = new origin >;

CommandList -> CommandList Command
                    <box$0 = c2(box$1, box$2, 0, 0, 0, 0)>
                    <pos$0 = pos$2>
                    <def$0 = defadd(def$1, def$2 )>
                    <env$1 = env$0 >
                    <env$2 = envadd(def$1, env$0 )>
                    <origin$1 = origin$0 >
                    <origin$2 = pos$1 > |

                 Command <box$0 = c(box$1, 0, 0, 0, 0)>
                         <pos$0 = pos$1 >
                         <def$0 = def$1 >
                         <env$1 = env$0 >
                         <origin$1 = origin$0 >;

Command -> _define _id ArgL Defn
              <box$0 = c2(box$3,box$4, 80, 2_1100, 2_0001, 2_0011)>
              <env$4 = envadd(def$3, env$0 )>
              <origin$4 = origin$0 >
              <pos$0 = origin$0 >
              <def$0 = do binding( val$4 )> |

           _id Arg1 Arg2
              <box$0 = c2(box$2, box$3, 80,
                                        2_100,
                                        0,
                                        2_011)>
              <env$2 = env$0 >
              <env$3 = env$0 >
              <pos$0 = do call(env$0, vals$2, vals$3,
                                              origin$0 )>
              <def$0 = 0 {vals$2 vals$3 } > |

           _line Arg1 Arg2
              <box$0 = c2(box$2, box$3, 80, 2_100,0,0)>
              <pos$0 = do line(vals$2,  vals$3, origin$0 )>
              <env$2 = env$0 >
              <env$3 = env$0 >
              <def$0 = 0 {vals$2 vals$3 }> |

           _colour Arg2
              <box$0 = c(box$2, 80, 2_10,0,0)>
              <pos$0 = do colour(vals$2,  origin$0 )>
              <env$2 = env$0 >
              <def$0 = 0 {vals$2 }> |

           _size Arg2
```

```
                <box$0 = c(box$2, 80, 2_10,0,0)>
                <pos$0 = do size(vals$2, origin$0 )>
                <env$2 = env$0 >
                <def$0 = 0 {vals$2 }> |

            _font Arg2
                <box$0 = c(box$2, 80, 2_10,0,0)>
                <pos$0 = do font(vals$2, origin$0 )>
                <env$2 = env$0 >
                <def$0 = 0 {vals$2 }> |

            _move Arg1 Arg2
                <box$0 = c2(box$2, box$3, 80, 2_100,0,0)>
                <pos$0 = do move(vals$2, vals$3,
                                 origin$0 )>
                <env$2 = env$0 >
                <env$3 = env$0 >
                <def$0 = 0 {vals$2 vals$3 }> |

            _text Arg1 Arg2
                <box$0 = c2(box$2, box$3, 80, 2_100,0,0)>
                <pos$0 = do text( vals$2, vals$3,
                                 origin$0 )>
                <def$0 = 0 {vals$2 vals$3 }>
                <env$2 = env$0 >
                <env$3 = env$0 >;

Arg1 -> _clb List _crb
            <box$0 = c(box$2, 50, 0,0,0)>
            <vals$0 = vals$2 >
            <env$2 = env$0 > |
            <vals$0 = 0>
            <box$0 = c0(default,0,0,0)>;


Arg2 -> _lb List _rb
            <box$0 = c(box$2, 50, 0,0,0)>
            <vals$0 = vals$2 >
            <env$2 = env$0 > |
            <vals$0 = 0 >
            <box$0 = c0(default,0,0,0)>;

ArgL -> _lb NL _rb
             <box$0 = c(box$2, 50, 0,0,0)>
             <def$0 = def$2 > |
             <def$0 = 0>
             <box$0 = c0(default,0,0,0)>;

NL -> NL _comma _id
          <box$0 = c(box$1, 30, 2_010,0,0)>
          <def$0 = def and bind(def$1 )> |
       _id
          <box$0 = c0(default, 0,0,0)>...
```

```
          <def$0 = do bind>;

List -> List _comma Item
          <box$0 =c2(box$1, box$3, 30, 2_010,0,0)>
          <vals$0 = addvals(vals$1, val$3 )>
          <env$1 = env$0 >
          <env$3 = env$0 > |

        Item
          <box$0 =c(box$1, default, 0,0,0)>
          <vals$0 = addval(val$1 )>
          <env$1 = env$0 >;

Defn -> _ass _slb CommandList _srb
          <box$0 =c(box$3, 60, 2_1000,0,0)>
          <val$0 = command ref(def$3 )>
          <origin$3 = copy origin(origin$0 )>
          <env$3 = env$0 > |

        _ass Item
          <box$0 =c(box$2, default, 2_10,0,0)>
          <env$2 = env$0 >
          <val$0 = val$2 >;

Item -> _lb Item _rb
          <box$0 =c(box$2, default, 0,0,0)>
          <val$0 = val$2 >
          <env$2 = env$0 >|

        _minus Item
          <box$0 =c(box$2, default, 0,0,0)>
          <val$0 = do negate(val$2 )>
          <env$2 = env$0 > |


        Item _times Item
          <box$0 =c2(box$1, box$3, default, 0,0,0)>
          <val$0 = do times(val$1,  val$3 )>
          <env$1 = env$0 >
          <env$3 = env$0 >|

        Item _div Item
          <box$0 =c2(box$1, box$3, default, 0,0,0)>
          <val$0 = do divide( val$1,  val$3 )>
          <env$1 = env$0 >
          <env$3 = env$0 > |

        Item _plus Item
          <box$0 =c2(box$1, box$3, default, 0,0,0)>
          <val$0 = do plus( val$1,  val$3 )>
          <env$1 = env$0 >
          <env$3 = env$0 > |
```

```
            Item _minus Item
               <box$0 =c2(box$1, box$3, default, 0,0,0)>
               <val$0 = do minus( val$1,  val$3 )>
               <env$1 = env$0 >
               <env$3 = env$0 > |

            _id
               <box$0 =c0(default, 0,0,0)>
               <val$0 = do name ref(env$0 )>|

            _int
               <box$0 =c0(default, 0,0,0)>
               <val$0 = do int>|

            _length Item
               <box$0 = c(box$2, default,2_10,0,0)>
               <env$2 = env$0 >
               <val$0 = do length(val$2 )> |

            _string
               <box$0 = c0(default, 0,0,0)>
               <val$0 = do string>;

  End of Productions

  Priorities  (_times, _div) (_plus, _minus);

End of Grammar
```

Appendix C

Attribute Grammar for Stick-Wright

```
Grammar Stic is

  Code [%include "stic.src"]

  Lexicals _cell, _abut, _equs, _ass, _lb, _rb, _slb, _srb, _clb, _crb,
           _lt, _gt, _arrow, _choice, _tilda, _hat, _comma, _colon,
           _semi, _stop, _dots, _minus, _plus, _times, _divide,
           _identifier, _integer [lex int],
           _true, _false, _and, _or, _not,
           _b, _g, _r, _bs, _gs, _rs, _bc, _gc, _rc, _gbx, _rbx,
           _pass, _dep, _enh, _bt, _gte, _rt,
           _rbc, _gbc, _rgcs, _rgcc;

  Synthesised
    Design (box, ports),
    CellList (box, def),
    Cell (box, def),
    Params (box),
    IList (box),
    PortSpec (box),
    OPorts (box),
    OList (box),
    List (box),
    Id (box),
    Ident (box, ports),
    QualS (box),
    ODir (box),
    Abut (box, ports, x, y),
    AbutBlock (box, ports, x, y),
    Row (box, ports, x),
    VRow (box, ports, y),
    Item (box, ports, x),
    OPar (box),
    Pars (box),
    OIter (box, x),
    Iter (box, x),
```

```
    Cond (box, ports),
    Syms (box, rot, mir),
    Sym (box, rot, mir),
    Range (box, x),
    OBar (box, ports, x),
    Expression (box, val),
    Condition (box, bool);


  Inherited
    Design (env),
    CellList (env),
    Cell (env),
    Params (env),
    IList (env),
    PortSpec (env),
    OPorts (env),
    OList (env),
    List (env),
    Id (env),
    Ident (env, origin),
    QualS (env),
    ODir (env),
    Abut (env),
    AbutBlock (env, origin),
    Row (env, origin),
    VRow (env, origin),
    Item (env, origin, dir),
    OPar (env),
    Pars (env),
    OIter (env),
    Iter (env),
    Cond (env, origin),
    Syms (env),
    Sym (env),
    Range (env),
    OBar (env, origin),
    Expression (env),
    Condition (env);

Productions

  Design -> CellList Ident
              <env$1 = 0>
              <env$2 = def$1>
              <ports$0  = ports$2>
              <origin$2 = top origin>
              <box$0 = c2(box$1, box$2, 0, 0, 0, 0)>;

  CellList -> CellList Cell
                <env$1 = env$0>
                <env$2 = def$1>
                <def$0 = add def(def$1, def$2)>
```

```
                <box$0 = c2(box$1, box$2, 0, 0, 0, 0)> |
               Cell
                <def$0 = def$1>
                <env$1 = 0>
                <box$0 = c(box$1, 0, 0, 0, 0)>;

Cell    -> _cell _identifier Params PortSpec _equs Abut _stop
            <def$0 = do binding( x$6, y$6, ports$6)>
            <env$6 = env$0>
            <box$0 = c3(box$3, box$4, box$6, default,
                                              2_1101100,
                                              2_0000001,
                                              2_0011010)>;

Params  -> _clb IList _crb
            <box$0 = c(box$2, default, 0, 0, 0)> |
            <box$0 = c0(default, 0, 0, 0)>;

IList -> IList _comma _identifier
          <box$0 = c(box$1, default, 2_010, 0, 0)> |
         _identifier
          <box$0 = c0(default, 0, 0, 0)>;

PortSpec -> _lb OPorts _rb
            <box$0 = c(box$2, default, 0, 0, 0)>;

OPorts -> OList _semi OList _semi OList _semi OList
          <box$0 = c4(box$1, box$3, box$5, box$7,
                      default, 0, 2_0101010, 0)>|
          <box$0 = c0(default, 0, 0, 0)>;

OList -> List
          <box$0 = c(box$1, 0, 0, 0, 0)>|
          <box$0 = c0(default, 0, 0, 0)>;

List -> List _comma Id
         <box$0 = c2(box$1, box$3, default, 2_010, 0, 0)> |
        Id
         <box$0 = c(box$1, default, 0, 0, 0)>;

Id -> _lb List _rb QualS OIter
       <box$0 = c3(box$2, box$4, box$5, default, 0, 0, 0)> |
      _identifier QualS OIter
       <box$0 = c2(box$2, box$3, default, 0, 0, 0)>;

QualS -> QualS _colon ODir _identifier
          <box$0 = c2(box$1, box$3, default, 0, 0, 0)>|
          <box$0 = c0(default, 0, 0, 0)>;


ODir -> _gt
         <box$0 = c0(default, 0, 0, 0)>|
        _lt
```

```
          <box$0 = c0(default, 0, 0, 0)>|
          <box$0 = c0(default, 0, 0, 0)>;

Abut -> _abut AbutBlock
          <ports$0 = ports$2>
          <x$0 = x$2> <y$0 = y$2>
          <env$2 = env$0>
          <origin$2 = new origin(x$2, y$2)>
          <box$0 = c(box$2, default, 2_10, 0, 0)>;

AbutBlock -> AbutBlock _semi Row
               <ports$0 = do Abut compose(ports$1, ports$3, y$1)>
               <x$0 = check length(x$1, x$3)>
               <y$0 = y$1 + 1>
               <origin$3 = do origin(origin$0, 0, y$1)>
               <origin$1 = origin$0>
               <env$1 = env$0>
               <env$3 = env$0>
               <box$0 = c2(box$1, box$3, default, 0, 2_010, 0)>|
             Row
               <origin$1 = origin$0>
               <ports$0 = ports$1>
               <env$1 = env$0>
               <x$0 = x$1> <y$0 = 1>
               <box$0 = c(box$1, default, 0, 0, 0)>;

Row -> Row _comma Item
        <ports$0 = do Row compose(ports$1, ports$3, x$1)>
        <origin$1 = origin$0>
        <origin$3 = do origin(origin$0, x$1, 0)>
        <env$1 = env$0>
        <env$3 = env$0>
        <dir$3 = 0>
        <x$0 = x$1 + x$3>
        <box$0 = c2(box$1, box$3, default, 2_010, 0, 0)>|
        Item
        <origin$1 = origin$0>
        <x$0 = x$1>
        <dir$1 = 0>
        <ports$0 = ports$1>
        <env$1 = env$0>
        <box$0 = c(box$1, default, 0, 0, 0)> ;


VRow -> VRow _comma Item
        <ports$0 = do Abut compose(ports$1, ports$3, y$1)>
        <origin$1 = origin$0>
        <origin$3 = do origin(origin$0, 0, y$1)>
        <env$1 = env$0>
        <env$3 = env$0>
        <dir$3 = 1>
        <y$0 = y$1 + x$3>
        <box$0 = c2(box$1, box$3, default, 2_010, 0, 0)>|
```

```
        Item
         <origin$1 = origin$0>
         <y$0 = x$1>
         <ports$0 = ports$1>
         <env$1 = env$0>
         <dir$1 = 1>
         <box$0 = c(box$1, default, 0, 0, 0)> ;

Item -> Ident Syms OIter
         <x$0 = x$3>
         <ports$0 = do port trans(ports$1, mir$2, rot$2,
                                  x$3, dir$0)>
         <origin$1 = do transforms(origin$0, mir$2, rot$2)>
         <env$1 = env$0>
         <env$3 = env$0>
         <box$0 = c3(box$1, box$2, box$3, default, 0, 0, 0)>;

Ident -> _lb Row _rb
          <origin$2 = do hor trans(origin$0, x$2)>
          <ports$0 = ports$2>
          <env$2 = env$0>
          <box$0 = c(box$2, default, 0, 0, 0)>|
         _lt VRow _gt
          <ports$0 = ports$2>
          <origin$2 = do ver trans(origin$0, y$2)>
          <env$2 = env$0>
          <box$0 = c(box$2, default, 0, 0, 0)>|
         _slb Cond _srb
          <ports$0 = ports$2>
          <env$2 = env$0>
          <origin$2 = origin$0>
          <box$0 = c(box$2, default, 0, 0, 0)>|
         _identifier OPar
          <ports$0 = do call( env$0, origin$0)>
          <box$0 = c(box$2, default, 0, 0, 0)> |
         _b
          <ports$0 = do b(origin$0)>
          <box$0 = c0(default, 0, 0, 0)> |
         _g
          <ports$0 = do g(origin$0)>
          <box$0 = c0(default, 0, 0, 0)> |

        . all the other primitives follow the above form .

         _rgcc
          <ports$0 = do rgcc(origin$0)>
          <box$0 = c0(default, 0, 0, 0)> |    {null cell}
          <ports$0 = do blank(origin$0)>
          <box$0 = c0(default, 0, 0, 0)>;


OPar  -> _clb Pars _crb
          <box$0 = c(box$2, default, 0, 0, 0)>|
```

```
          <box$0 = c0(default, 0, 0, 0)>;

Pars  -> Pars _comma Expression
          <box$0 = c2(box$1, box$3, default, 2_010, 0, 0)>|
         Expression
          <box$0 = c(box$1, default, 0, 0, 0)>;

OIter -> _slb Iter _srb
          <env$2 = env$0>
          <x$0 = x$2>
          <box$0 = c(box$2, default, 0, 0, 0)>|
          <x$0 = 1>
          <box$0 = c0(default, 0, 0, 0)>;

Iter  -> _identifier _ass Range
          <box$0 = c(box$3, default, 2_110, 0, 0)>
          <x$0 = x$3>|
         Range
          <env$1 = env$0>
          <x$0 = x$1>
          <box$0 = c(box$1, default, 0, 0, 0)>;

Cond -> Condition _arrow Item OBar
          <origin$3 = origin$0>
          <origin$4 = origin$0>
          <env$1 = env$0>
          <env$3 = env$0>
          <env$4 = env$0>
          <dir$3 = 0>
          <ports$0 = do condition(bool$1, ports$3, ports$4,
                                  x$3, x$4)>
          <box$0 = c3(box$1, box$3, box$4, default, 2_1100, 0, 0)>;


Syms -> Syms Sym
         <box$0 = c2(box$1, box$2, default, 0, 0, 0)>
         <rot$0 = rot$1 + rot$2>
         <mir$0 = mir$1 + mir$2>
         |
         <mir$0 = 0>
         <rot$0 = 0>
         <box$0 = c0(default, 0, 0, 0)>;

Sym  -> _tilda
         <mir$0 = 1>
         <rot$0 = 0>
         <box$0 = c0(default, 0, 0, 0)>|
        _hat
         <rot$0 = 1>
         <mir$0 = 0>
         <box$0 = c0(default, 0, 0, 0)>;

Range -> Expression _dots Expression
```

```
            <env$1 = env$0> <env$3 = env$0>
            <x$0 = | val$3 - val$1 | + 1>
            <box$0 = c2(box$1, box$3, default, 2_110, 0, 0)>|
           Expression
            <x$0 = val$1>
            <env$1 = env$0>
            <box$0 = c(box$1, default, 0, 0, 0)>;

OBar   -> _choice Item
            <ports$0 = ports$2>
            <env$2 = env$0>
            <dir$2 = 0>
            <x$0 = x$2>
            <origin$2 = origin$0>
            <box$0 = c(box$2, default, 0, 0, 0)>|
            <ports$0 = do blank(origin$0)>
            <box$0 = c0(default, 2_10, 0, 0)>;


Expression -> _minus Expression
               <env$2 = env$0>
               <val$0 = -val$2>
               <box$0 = c(box$2, default, 0, 0, 0)>|
              _lb Expression _rb
               <env$2 = env$0>
               <val$0 = val$2>
               <box$0 = c(box$2, default, 0, 0, 0)>|
              Expression _times Expression
               <env$1 = env$0> <env$3 = env$0>
               <val$0 = val$1 * val$3>
               <box$0 = c2(box$1, box$3, default, 0, 0, 0)>|
              Expression _divide Expression
               <env$1 = env$0> <env$3 = env$0>
               <val$0 = val$1 // val$3>
               <box$0 = c2(box$1, box$3, default, 0, 0, 0)>|
              Expression _plus Expression
               <env$1 = env$0> <env$3 = env$0>
               <val$0 = val$1 + val$3>
               <box$0 = c2(box$1, box$3, default, 0, 0, 0)>|
              Expression _minus Expression
               <env$1 = env$0> <env$3 = env$0>
               <val$0 = val$1 - val$3>
               <box$0 = c2(box$1, box$3, default, 0, 0, 0)>|
              _identifier
               <val$0 = do name ref(env$0)>
               <box$0 = c0(default, 0, 0, 0)>|
              _integer
               <val$0 = do integer>
               <box$0 = c0(default, 0, 0, 0)>;

Condition -> _lb Condition _rb
              <env$2 = env$0>
              <bool$0 = bool$2>
```

```
 <box$0 = c(box$2, default, 0, 0, 0)>|
_not Condition
 <env$2 = env$0>
 <bool$0 = \bool$2>
 <box$0 = c(box$2, default, 0, 0, 0)>|
Condition _and Condition
 <env$1 = env$0> <env$3 = env$0>
 <bool$0 = bool$1 & bool$3>
 <box$0 = c2(box$1, box$3, default, 0, 0, 0)>|
Condition _or Condition
  <env$1 = env$0> <env$3 = env$0>
  <bool$0 = bool$1 ! bool$3>
  <box$0 = c2(box$1, box$3, default, 0, 0, 0)>|
Expression _equs Expression
 <env$1 = env$0> <env$3 = env$0>
 <bool$0 = do equs(val$1, val$3)>
 <box$0 = c2(box$1, box$3, default, 0, 0, 0)>|
Expression _lt Expression
 <env$1 = env$0> <env$3 = env$0>
 <bool$0 = do lt(val$1, val$3)>
 <box$0 = c2(box$1, box$3, default, 0, 0, 0)>|
Expression _gt Expression
 <env$1 = env$0> <env$3 = env$0>
 <bool$0 = do gt(val$1, val$3)>
 <box$0 = c2(box$1, box$3, default, 0, 0, 0)>|
_true
 <bool$0 = -1>
 <box$0 = c0(default, 0, 0, 0)>|
_false
 <bool$0 = 0>
 <box$0 = c0(default, 0, 0, 0)>;

  End of Productions

  Priorities (_times, _divide) (_plus, _minus);

End of Grammar
```