COMPUTER SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES
DEPARTMENT OF ELECTRICAL ENGINEERING
STANFORD UNIVERSITY. STANFORD. CA 94305

SEL 79-027

ADLIB USER'S MANUAL

by

Dwight D. Hill

Technical Report No. 177

August 1979

This work was supported by the Joint Services Electronics Program under Contract DAAG29-79-C-0047.

SEL 79-027

ADLIB USER'S MANUAL

Dwight D. Hill

Technical Report No. 177

August 1979

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

---

*This work was supported by the Joint Services Electronics Program under Contract DAAG29-79-c-0047

ADLIB USER'S MANUAL

Dwight D. Hill

Technical Report No. 177

August 1979

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

ABSTRACT

ADLIB (A Design Language for Indicating Behavior> is a new computer design language recently developed at Stanford. ADLIB is a superset of PASCAL with special facilites for concurrency and int erprocess communication. It is normally used under the SABLE simulation system.

INDEX TERMS : simulation, PASCAL, hardware description languages, SABLE, SDL

Introduction

SABLE is a design automation system currently being developed at Stanford University to support structured, multi-level simulation of computer designs. SABLE stands for Structure And Behavior Linking Environment, because it joins information about the interconnectivity of components with algorithmetic specifications of their behavior. The user expresses interconnectivity via SDL (Structural Design Language) [VCW77], which has facilities for defining multiple levels of physical hierarchy. Component behavior is specified in ADLIB (A Design Language for Indicating Behavior), which is a superset of the language PASCAL [JK74]. ADLIB was designed to simplify the description of commonly used computer components and to be compatible with SDL and SABLE. ADLIB was never intended to be a programming language completely by itself.

This manual is divided into three parts: an introduction to ADLIB with a informal description of the basic features; a more detailed discussion of the structure of an ADLIB program, including scoping rule and contour models; and a summary of the keywords and syntax of the language. Because documentation is widely available, this manual will not repeat the detailed features of PASCAL. However, the basic aspects of PASCAL will be described briefly, so that readers familiar with other high level languages should be able to follow the discussion without too much difficulty.

The author would like to express his thanks to William vanCleemput, Warren Cory, Eric Slutz, Tom Blank, Ann Beetem, Bob Dutton, John Hennessey, and all the other, people who have contributed to ADLIB.

TABLE OF CONTENTS

CHAPTER 1
BASICS OF ADLIB

1.1 PURPOSE OF ADLIB

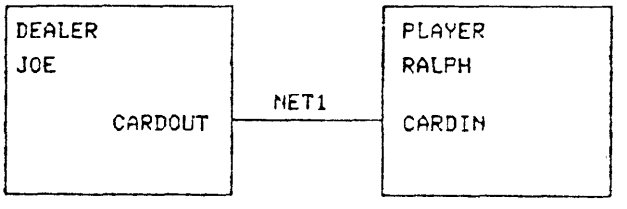
The purpose of an ADLIB description is to define the behavior of one or more types of computer components. The SABLE system then combines these with information that specifies the number of components used, and the way they are connected. This topological information is expressed in SDL. For convenience the user may generate the SDL automatically via an interactive graphical structure editor called SUDS2 [US79].

In ADLIB, the code that defines the behavior of one type of component is called a comptype. There is no way of telling from an ADLIB source how many components of each comptype, if any, will be used in a design. Each comptype written in ADLIB is a specification of the input to output function of one type of component. Essentially all information that passes through a component must go through well defined I/O interfaces called "nets." SABLE later connects these nets to the nets of other components as directed by the user via SDL.

Before we enter into any explanations of ADLIB and SABLE, it might be helpful to give a small, useless but complete example. We will define a tiny system consisting of a dealer and a player. the dealer sends random integers to the player, who just receives them and writes the results on the terminal. The ADLIB code for this looks like:

```
PROGRAM exmpl;

NETTYPE
intnet =integer;

#include rpaks.dcl !include a file of routine declarations

COMPTYPE dealer;
OUTWARD
cardout : intnet;
VAR
i : integer;
BEGIN
WHILE true  DO BEGIN
        ASSIGN rndint(1,13) TO cardout;
        WAITFOR true DELAY 1.0;
        END;
END;

COMPTYPE player;
INWARD
cardin : intnet;
BEGIN
WHILE true DO BEGIN
        WAITFOR true CHECK cardin;
        writeln(tty,cardin);
        END;
END;

BEGIN
END,
```

The structure of this system is shown below:



The SDL code to used describe the interconnectivity of this system is:

```
NAME: TEST;
TYFES :dealer,player;
dealer : joe;
player : ralph;
END;
NETSEGMENT;
net 1 = joe.cardout,ralph.cardin;
ENDNETS;
ENDC;
CEND;
```

When this code is compiled and executed, the result is a never ending stream of random integers between 1 and 13 (inclusive) directed to the terminal. If the example makes sense, fine. If not, don't worry. The remander of this paper will explain and elaborate everything.

1.2 INTRODUCTION TO THE NEW ADLIB CONSTRUCTS

Because ADLIB is a superset of PASCAL, it includes all of the PASCAL control statements. For readers not familiar with these statements they are summarized here.

1.2.1 Review Of PASCAL Constructs

1. IF <boolean expr> THEN <stmt1> ELSE <stmt2>

   which chooses between two alternative statements;

2. CASE <expr> OF <value1>:<stmt1>
   <value2>:<stmt2>
   ...
   END

   which selects one of an arbitrary number of statements, (similar to, but more powerful than a "switch" or "computed goto");

3. WHILE <boolean expr> DO <stmt>

   which iterates a statement zero or more times;

4. REPEAT <stmt> UNTIL <boolean expr>

   which iterates a statement one or more times;

5. FOR <variable>:=<expr 1> TO<expr2> DO <stmt >

   which repetitively executes a statement as <variable> ranges from <expr1> t o <expr2>. (Similar to a FORTRAN DO loop.)

6. GOT0 <label>

   which transfers control to <label> unconditionally%.

1.2.2 Additional ADLIB Constructs

The above six constructs are useful for defining tfie algorithm incorporated wit hin a component , but are not adequate for describing inter-component control and data flow. Therefore, the following new constructs have been added to ADLIB:

1.2.2.1 ASSIGN <expr> TO <net name> <timing clause>

Assign evaluates <expr> and stores the result away in a hidden area. At a later time, this value is retrieved and assigned to the specified net. Time delays may be specified in several ways depending on the nature of the circuit (synchronous or asynchronous) and the objectives of its designer. The simplest way is to define a delay directly, as for example:

ASSIGN true TO out DELAY 15.3;

Fifteen and three tenths simulated time units after this statement is executed, the net "0ut" will be updated to the value “true”. Time delays need not be constants, any real expression may be used. For example, if two parallel paths exist to the same outward connection, and either one is sufficient to drive it, then we could define the component ' s behav i or as:

*This construct is currently out of fashion.

```
ASSIGN result TO out DELAY min(delay_1,delay_2)
```

("Min" is a function that returns the minimum of its arguments.)

The expression in an ASSIGN statement may contain function calls. For example, in order to describe a signal generator, it is convenient to write:

```
ASSIGN sin(time*frequency) TO signalout
```

This statement illustrates two other points as well. In ADLIB, the variable "time" always contains the current value of the simulation time. When the simulation begins, it is equal to 0.0. User assignment to "time" results in a compilation error message. Also, this statement does not contain an explicit DELAY clause. The ADLIB compiler therefore treats it as if DELAY 0.0 were specified. At first glance, zero propagation delay times may seen confusing, unrealistic, and potentially hazardous. However, because of the runtime organization of SABLE, this operation is unambiguous and useful. During simulation, SABLE cycles between the execution of component's behavior descriptions and the updating of the nets connecting them; First, all components are allowed to execute, then all nets are updated, then all components are allowed to execute again, etc. One iteration of this cycle constitutes one event. It may happen that several events occur sequentially, but at the same simulated time. If one or more components assign to a set of nets with a DELAY of 0.0, then all these updates will appear to occur simultaneously.

No hazards or races are introduced by allowing zero propagation delay, and there are several applications where it is in fact, necessary and appropriate. For example, a designer may prefer to treat combinational logic as operating with zero time delay, to contrast it with the sequential circuitry. As an extreme example of this, consider

a system implemented with relays. We wish to express the idea that a voltage propagates through the contacts of a relay immeasurably faster (measured in nano-seconds) than the speed at which the armature moves (measured in milli-seconds). To describe a relay which operates as a single pole, dduble throw switch (like a one bit multiplexor), we could write:

```
IF armature-positicn = UP THEN ASSIGN input1 TO out
     ELSE ASSIGN input2 TO out
```

In the above example, the exact speed of propagat ion is incalculable and irrelevant (in fact, it would probably be lost to round-off error>. On the opposite extreme are circuits whose output values must be available art preci rely controlled instants, i.e. synchronous circuits. For example, most micro-controllers operate at a precisely constant speed independent of the micro-instruction mix (this is not generally true of macro-instructions), Such controllers and any circuitry directly connected with them are most conveniently defined in ADLIB with the use of a CLOCK and the SYNC primitive. An ADLIB clock may be thought of as a function that maps simulation time into positive integers. At time 0 all clocks have value 0. As simulation time progresses, the clocks run through their phases repetitively: 0,1,2,3,0,1,2,3,0,1,2,3... etc. for a four phase clock. The period of repetition is the parameter value specified by the user in the clock definition statement. The value of clock "clk," defined as:

```
CLOCK clk(4.0,4);
```

is shown in figure 1.

By use of the SYNC operator, the user can synchronize an operation with a particular leading edge of a clock. For example, a micro-controller might have to have several control 1 ines ready at precisely the leading edge of the number one phase of clock "micro_clk". This could be written as:

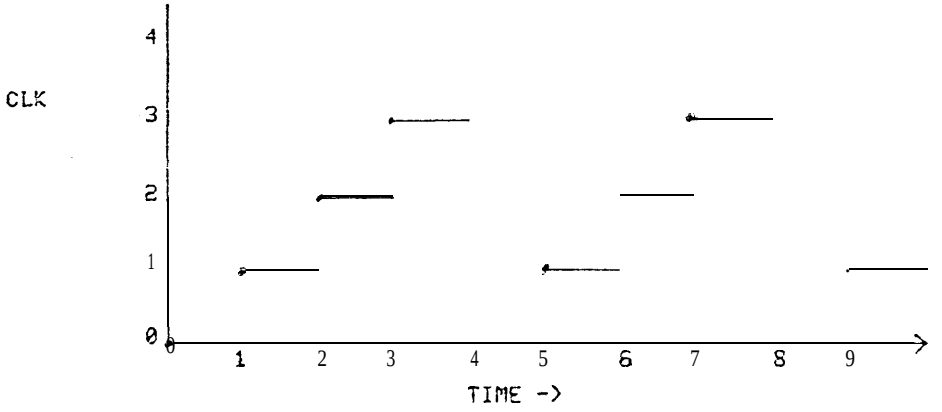


FIGURE 1: VALUE OF CLK(4.0,4)

```
r := microstore[micro_ip];
micro-ip := micro-ip + 1;
ASSIGN r.carry TO line1 SYNC micro-elk PHASE 1;
ASSIGN r.shift0 TO line2 SYNC micro-elk PHASE 1;
ASSIGN r.shift1 TO line3 SYNC micro-elk PHASE 1;
ASSIGN r.shift2 TO line4 SYNC micro-elk PHASE 1;
ASSIGN r.clear TO line5 SYNC micro-elk PHASE 1;
 (*etc.*)
```

All of the above ASSIGN statements will be effected at precisely the same simulated time.

The user may specify any number of independent CLOCKS, each with their own periods and numbers of phases. Unlike some other simulation environments, clocks do not consume any computation resources themselves: only when and if a component accesses them is any calculation performed. The user may mark one of the clocks as being the default. This saves him or her from writing the clock's name in every sync clause. Also, if no phase is specified, the compiler assumes that phase 0 is intended. It is therefore quite convenient to describe systems that maintain 3 single universal clock, such as a pipelined multiplier that keeps each stage in lock step with the others.

1.2.2.2 WAITFOR <boolean expr> <control clause>

WAITFOR causes execution of a component's code to stop, and does not allow it to continue until <boolean expr> evaluates to true. The <control clause> may come in one of two forms. First, a <timing clause> may be used, just like the timing clause in an ASSIGN statement. If a delay clause if used, the <boolean expr> is reevaluated periodically at the period specified in the delay clause. For example:

```
WAITFOR current>0.001  DELAY sample-period;
```

This statement would check the value of "current" every "sample-period" time units, until it exceeded one milliamp.

If SYNC is specified, the <boolean expr> is reevaluated each time the specified clock goes through the specified phase. For example:

```
WAITFOR acknowledge=1 SYNC bus-clock PHASE 4;
```

This statement would not allow execution to continue until the net "acknowledge" was equal to 1 on the leading edge of the fourth phase of clock "bus-clock".

Alternatively, a control clause may take the form of a list of nets that the component is to be sensitized to. This format is called a "check list", because whenever one of the nets mentioned in it is updated, the boolean expression is rechecked. By this means, it is easy and efficient to describe asynchronous machines driven by the the nets to which they are connected. For example:

```
WAITFOR data-rdy = 1 CHECK data-rdy
```

This statement would put the component into a passive state until the

net "data_rdy" was updated to the value 1. No simulation resources are consumed while the component is idle (there is no "busy waiting"). In particular, if other components ASSIGN to data-rdy whatever value that it already contains, the expression "data-rdy = 1" is not reevaluated. This is because SABLE automatically deletes all such null updates.

1.2.2.3 Sensitize, Desensitize, and Detach

Taken collectively, these provide a facility for direct control of the operation of a component. They operate in a way that is similar to WAITFOR, but at a lower level and somewhat more efficiently. Sensitize and desensitize are predefined procedures make a component receptive or immune to changes on its inward nets. These procedures are always used in conjunction with the DETACH operator, which causes execution of a component to stop until one or more of the nets to which it is sensitive is updated. When an update on a sensitized net occurs, the component will be awakened.

Because of the flexibility of the WAITFOR construct, it is difficult to think of an application where DETACH is really more convenient, and not merely more efficient. However, to illustrate its use, we shall use it to describe a finite state machine that recognizes the bit strings consisting of 1's and 0's. The strings must match a regular expression that begins and ends with 1, and where any 0 must be preceeded and followed by at least one 1 (example taken from [KZ70]). This machine is stimulated by a net called "input-line", which contains a data element "d" and a strobe field "s". (In order to drive this machine, it is necessary to put the data value in the "d" field, and to update the "s" field.) In ADLIB, one way to define the automaton is:

```
sensitize(input_line);

1: detach; (*initial state*)
   if input-1ine.d = 0 then goto 3;

2: (*accepting state*)
   writeln(output,'accept');
   detach;
   if input-1ine.d = 0 then goto 1;
   goto 2;

3: detach; (* terminal state*)
   goto 3;
```

When combined with the facilities available in PASCAL, the above primitives are adequate to encode almost any conceivable function among a component's nets. However, before a design language will be used, it must be more than adequate, it must be convenient. Therefore ADLIB also incorporates the concept of "subprocesses" to facilitate encoding the behavior of many common computer activities. Each subprocess performs a single function and runs independently of the main body of the comptype. There are two types of subprocesses, upon and transmit.

1.2.2.4 UPON <boolean expr> <check list> DO <stmt>

Upon is used to define a set of activities to be performed independently of the main activities of the component. Whenever a net in <check list> is updated, <boolean expr> is reevaluated. If it is true, then <stmt> is executed. For example:

```
interrupt : UPON  (interrupt.priority > current)
        CHECK interrupt DO
              BEGIN
              push(machine_state);
              service_interrupt;
              pop(machine_state);
              END;
```

This code would check the priority level whenever the interrupt net was updated, and service the interrupt when necessary.

1.2.2.5 TRANSMIT <expr> TO <net> <timing clause?

TRANSMIT is more specialized than UPON. A transmit subprocess reevaluates <expr> whenever one or more of the nets in <net list> is changed. The result is then assigned to <net name> at the time specified by <timing clause>. Transmit is very convenient for describing combinational circuitry. For example, a simple NAND gate can be described with:

```
nand:TRANSMIT NOT(a AND b) TO c CHECK a,b DELAY 15.0;
```

1.2.2.6 Inhibit And Permit

The name given to a subprocess may be used to control it by means of the procedures "permit" and "inhibit". All subprocesses are initially inhibited, which means that no external stimulus can activate them. The main body of a component may then permit some or all of the subprocesses to run, at which point they are ready to respond to stimuli. Subprocesses may be inhibited at any time, which returns them to their initial, inactive state. For example, a computer may protect a critical region with:

```
inhibit(interrupt);
write_to(shared_data);
permit(interrupt);
```

1.3 DESIGNING AN ADLIB PROGRAM

To illustrate how ADLIB is used, we shall "design" a small system that plays Black jack with itself. This was inspired by a DDL design found in [DDL75], but is somewhat more complex and complete. The ADLIB system consists of one or more dealers and one or more players. Nets shall be used to represent the flow of cards from the dealers to the players, and to coordinate their activities. To begin our design

process, we first consider what data must be transmitted (suit and rank> and what control information is needed (player is waiting for card, dealer is waiting for player, etc.) We develop three types of nets (nettype's) that define the structure of the nets which carry information between components, and associate an interpretation with them. The design at this point might be encoded as shown in figure 2. The nettypes shown fall into two categories: structured nettypes, such as "card_bus" and simple ones, such as "display-lights" and "control_line". Structured nettypes are most useful when several pieces of information are logically affiliated but need to be updated and examined independently.

```
PROGRAM cardgame;
TYPE
suit-type = (clubs,hearts,diamonds,spades);
rank-type = (ace,two,three,four,five,six,seven,
   eight,nine,ten,jack,queen,king);
NETTYPE
card_bus = RECORD
   suit : suit-type;
   rank : rank-type;
   END;
display-lights = (hit,stand,broke);
control_line = (card_rdy,card_accepted);

COMPTYPE dealer; (* deals out cards*)
   BEGIN
   (* not yet designed*)
   END;

COMPTYPE player; (*accepts cards, stands or goes broke*)
   BEGIN
   (* not yet designed*)
   END;
BEGIN
END.
```

Figure 2 - Outline of Blackjack System

The code in figure 2 specifies three net types but no control protocol. We decide to use one control line between each player and the dealer that serves it, and to alternate the value of this net between "card_rdy" and "card-accepted". The synchronization mechanism can then be expressed by the code fragments shown in figure 3. The first waitfor statement shown causes the player to wait until the card is ready, and

the second causes the dealer to wait until the player has decided what to do.

```
(* player's code *)
WAITFOR cntrl = cardrdy CHECK cntrl;
(* accept card*)
ASSIGN card-accepted TO cntrl;
(* process card, go broke, hit or stand*)

(* dealer's code *)
(* generate next card *)
(* assign nextcard to cardbus *)
ASSIGN card-rdy TO;
WAITFOR cntrl = cardaccepted CHECK cntrl;
```

Figure 3 Code fragments for Coordination

We are now ready to specify the algorithms used by the players and dealers, For the purpose of this discussion, the player just accepts cards until it reaches its limit, treating aces as 1 or 11 points as needed. The ADLIB code for the player is shown in figure 4, and will be referred to later.

```
COMPTYPE player;

(* declare the "terminals" of this component*)
INWARD
card : card-bus;
OUTWARD
lights : display-lights;
EXTERNAL
cntrl : control-line;

(*declare the storage needed by this component++)
VAR
score :0..27;
holding-ace : boolean;
BEGIN
WHILE true DO BEGIN
holding_ace := false;
score := 0;
REPEAT
      REPEAT
              ASSIGN hit TO lights;
              ASSIGN card-accepted TO cntrl;
              WAITFOR cntrl=card_rdy CHECK cnfrl;
              IF card.rank < jack THEN score :=
                       score + ord(card.rank) + 1
               (ford returns an integer in 0..13*)
               ELSE score := score +10;
              IF (card.rank=ace) AND
               (NOT holding-ace)
                       THEN BEGIN
                       score := score + 10;
                       holding_ace:=true END;
              UNTIL score >= upper-limit;
      IF (score > 21) AND holding-ace THEN BEGIN
              score := score - 10;
              holding-ace := false END;
      UNTIL score >= upper limit;
IF score <= 21 THEN ASSIGN stand TO lights
      ELSE ASSIGN broke TO 1 ights;
END;
END;
```

Figure 4: Definition of Comptype "Player"

In order to see if comptype "player" works properly, it is necessary t 0 develop a "dealer" comptype to drive it. There are several possible ways to do this, just as there are several ways to test a new piece of hardware. In ADLIB, it is easy to write a comptype that talks with the terminal for interactive testing. Another possibility is to use a pseudo-random number generator that will choose cards from an infinite deck. A package of such generator routines called RNDPAK is available to ADLIB users. Finally, the designer can write a comptype that reads the test data from a file. Each of these approaches have

their own merits at different phases in the design process. Normally, interactive testing would be used for initial debugging, large numbers of random inputs for extensive testing, and prerecorded, specially selected values for production.

Having specified the behavior of comptype "player" and "dealer", we are free to use as many of each as we wish in our design. The structure of one possible cardgame is shown in figure 5.

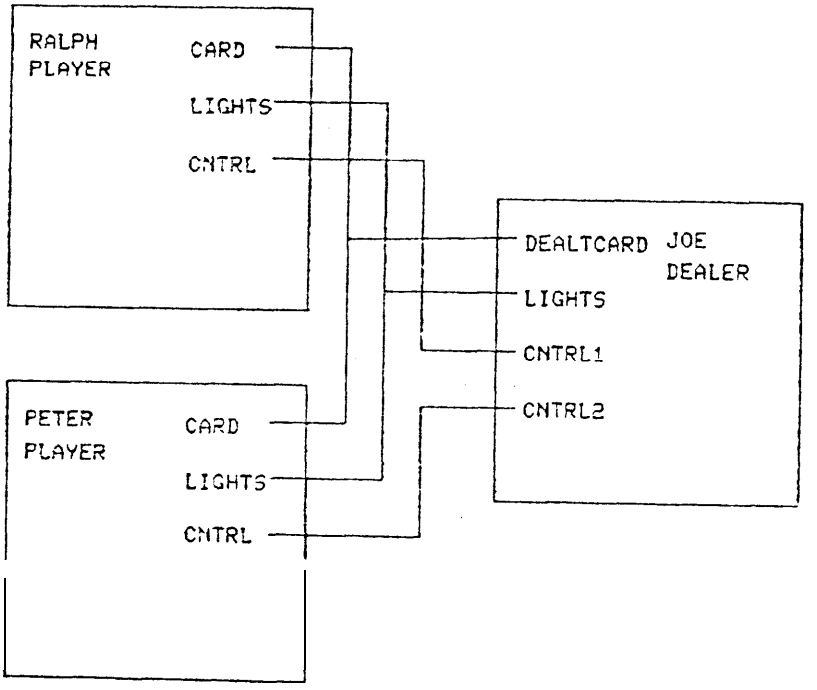


FIGURE 5 : STRUCTURE OF A SIMPLE CARDGAME

CHAPTER 2
TYPES, NETTYPES AND TYPE CHECKING

2.1 WHAT DOES "STRONGLY TYPED" MEAN?

Like its base language PASCAL, ADLIB is said to be a "strongly typed" language [OEI78]. This means that each piece of data, each function and all parameters must each be declared with exactly one type. Whenever a variable or net is used, its use must be type compatible with its declaration. In a well-written program, the type of a variable defines the narrowest possible range of values that it may attain. While beginning programmers may think that type checking is an arbitrary restriction on their programming style, more experienced designers can usually put it to very good use. Typing is essentially a way for the designer to express his or her intentions about the way in which a piece of data should be used. The compiler can then automatically detect when those intentions are violated, which usually implies an error. In languages that are not typeful, it is often easy to treat character strings as reals, or integers as pointers. Even when this is done intentionally, it is very difficult to read, understand and maintain the resulting program. And when it is done unintentionally, chaos can result. For a discussion of PASCAL typing and program reliability the reader is referred to [WN75].

2.2 TYPE CHECKING OF NETS

The primary type checking mechanism in ADLIB is the use of nettypes for defining the intended interconnection mechanism between components.

In SABLE, the nets that connect components must be declared with identical nettypes at each end. If this is not so, an error message is printed++. This nettype checking is considerably more thorough than checks found in most register transfer language's (RTL's), where it is only necessary that the number of bits must match. To illustrate this difference, consider a component that produces two BCD (binary coded decimal) digits and another that accepts eight bits of binary data. Most RTL's would allow them to be directly connected, since 8 bits = 8 bits. Even simulation might not detect this error if the test set did not happen to include any values greater than 9. But the ADLIB - SABLE environment would detect the mismatch, since "BCD" is not type compatible with "binary". Further examples of type checks will be given later in this section.

But types should not be viewed just as a restriction. Compared with other languages, ADLIB (like PASCAL) offers a wealth of useful new types to choose from. The next section illustrates a few of the ways that a designer can take advantage of them to reduce errors and improve readability.

2.3 DATA TYPES AVAILABLE IN ADLIB

Because ADLIB is a superset of PASCAL, it inherits all of the PASCAL type construction mechanisms. For the benefit of readers not familiar with PASCAL, these are listed here:

*Unless a special TRANSLATOR has been provided. TRANSLATORS provide the ability to do multi-level simulation at the expense of some type-checking security and some loss of data precision. For a discussion of their use, see [HDD79-1].

2.3.1 Boolean

Boolean variables can attain the values true or false. (Similar to FORTRAN's LOGICAL.) For example if "strobe" were declared to be of type boolean we could write:

```
strobe := data-rdy AND (bus-clock = 3)
```

2.3.2 Integer

I.e. -15, or 1024.

2.3.3 Real

I.e. 3.14159 or 6.023e24

2.3.4 Enumerated Types

These allow a user to enumerate (list out) all the possible values of a piece of data. For example:

```
logic-level = (low, high, unknown, high-impedance)
```

We have already seen several examples of enumerated types in the cardgame program. Enumerated types are also useful for describing the instruct ion sets of machines, such as the INTEL 8008 [I73]:

```
TYPE
instruct-kind = (lrr,lrm,lmr,lri,lmi,
   inr,dcr,adr,adm, (*etc*) );
```

In ADLIB, one can use a CASE statement to describe the execution of a machine instruct ion in a format very similar to ISP [BCG71]. For exampl e:

```
FUNCTION decode : instruct_kind;
(* code omitted for brevity*)

BEGIN (*main body of program*)
CASE decode OF
   lrr  : (* load reg to reg*)
          BEGIN (* code omitted *) END;
   ret  : (* return from subroutine++>
          BEGIN (* code omitted*) END;
   (*etc*)
   END;
```

2.3.5 Subranges

These specify that only part of a range of values is acceptable, for example:

register-number = 0..7

specifies not only that variables of type register-number are integers, but also that they must lie between 0 and 7 (inclusive). Assigning a value to a subrange variable that is outside its range is automatically detected. For example, in the ADLIB blackjack machine the variable "score" was declared to range over the values 0 to 27, the high value being equal to 16 (the highest possible score before standing) + 11 (value of an ace). By contrast, the DDL version merely declared the score to be a five bit register. The ADLIB approach has two advantages. First, it allows the designer to defer any decision on the representation of data in the early stages of design. Second, and more importantly, it encodes more information: e.g. the fact that the score can never exceed 27. Although this is not too critical here, it is easy to visualize applications where the range of data that a register holds, if known, can be used to improve the design. For example, nine ACLIB blackjack scores could safely be added in an 8-bit alu, since we know the total cannot exceed 243. On a more practical level, it might be useful to know not only that memory addresses in a DEC 10 are 18 bits

long, but also that they range from 16 to 262144, (since the first 16 addresses refer to registers).

2.3.6 Arrays .

ARRAY is similar to DIMENSION in FORTRAN. For example:

```
memory = ARRAY[0..10231 OF integer
```

2.3.7 Records

Records are useful for grouping related data, as for example:

```
complex = RECORD
        real_part, imag_part : real;
        END;
```

Most register transfer languages, including DDL, provide mechanisms for making several names equivalent references to the same piece of data. The usual example of this is an instruction register, where one of the bits is given a mnemonic name such as "I" (for Indirect) in addition to being IR[0]. This can make parts of a program more readable, but can also lead to confusion when a mnemonic is referenced for the first time several pages away from its declaration. The strategy adopted by ADLIB is to use "variant" records for this purpose. A variant record is essentially a single data area that may have several different data structure "templates" applied to it. As an example of this, consider the four ways that one can look at an HP 21 16 instruct ion, as discussed in the machine manual [HP]. In ADLIB, these alternative views would be encoded as:

```
TYPE
instr-variant = (whole,memory_ref,register_ref,i_o);
VAR
ir : RECORD CASE instr-variant OF
        whole : (ARRAY[0..151 OF bit);
        memory-ref : (indirect : bit;
                mem_instr : ARRAY[0..4] OF bit;
                zero : bit;
                mem_addr : ARRAY[0..9] OF bit);
        reg_ref : (group : ARRAY[0..3] OF bit;
                micro : ARRAY[0..111 OF bit);
        i_o : (io_group : ARRAY[0..3] OF bit;
                io_instr : ARRAY[0..5] OF bit;
                select : ARRAY[0..5] OF bit;
        END;
```

This record informs the reader (and the compiler> that the instruct ion register "ir" may be viewed in four different ways, but is still in fact just one register 16 bits long. (Note that the total number of bits in each variant is 16.) Access to ir can then be performed using the mnemonic fields, as for example:

```
ir.whole := data-bus;
```

or

```
IF ir.indirect = 1 THEN cycle := fetch:
```

Now the fields are closely associated with the register, and the reader is (hopefully> less likely to misinterpret them.

2.3.8 Sets

A SET is an area of storage that may contain from 0 to all of its members i . e. a powerset. Textually, sets are delimited by "[" and "]", and facilties are provided for set intersect ion (AND), union (OR), difference (-) membership (IN), equality ("="), size comparison ("<" and ">"), Since set s are normally packed into machine words, these operations usually run very quickly. Sets are convenient for grouping related symbo 1s, both visually for the reader and logically for simulation. For exampl e, in the 8008 we can express certain facts in machine readable form that are normally only shown on the data sheets,

such as:

```
index-instructs:= [lrr,lrm,lmr,lri,lmi,inr,dcr];
one_cycle_alu:=[adr,acr,sur,sbr,ndr,xrr,orr,cpr];
```

As an example of the use of set operators, consider the code that describes the timing of part of the execution cycle. It might contain:

```
IF instruct ion IN one-cycle-alu THEN
WAITFOR SYNC PHASE 1;
```

2.3.9 Files

The ADLIB user may declare various types of FILEs to match the data to be stored in them. Storing and retrieving that data can then be accomplished very efficiently. For example, it is easy to describe a core image as:

```
core-image : FILE OF integer;
```

For convenience, special facilties are provided for reading and writing files of text.

2.3.10 Pointers

PASCAL, (and therefore ADLIB) provides two independent areas for storing data, the ordinary stack and a heap. The heap is accessed only via special pointer variables, which may in turn point to other pointers, etc. This makes it convenient and efficient to develop complex data structures. Pointers are denoted by the up arrow "↑". For example, the data structures for describing a virtual memory system might look like:

```
TYPE
page = ARRAY[0..511] OF integer;
page-ptr = ↑page;(* "↑" means "pointer to"*)
page-table = ARRAY [0..255] OF RECORD
  logical-address : 0..64255;
  is_incore : boolean;
  memory-ref : page-ptr;
  END;
```

New elements are added to the heap by means of the "new" procedure. Using this, part of the code to describe memory management might be written:

```
VAR pm : pagemap;

IF NOT pm[high_bits].is_incore THEN new(pm.memory_ref);
```

This would allocate a new page of memory from the heap if the "is_incore" flag of page-map "pm" were false.

Whereas the above types are part of both PASCAL and ADLIB, the following two are available only in ADLIB.

2.3.11 Bit

This is a predeclared subrange of integer. It may range over the values 0..1.

2.3.12 Register

This is a predeclared type that is useful for RTL descriptions. Many special routines are provided for manipulating variables of type register, such as exclusive or, rotate, etc. Arithmetic may be performed on registers in one's complement, two's complement, sign magnitude, and unsigned formats.

2.3.13 Another Example: RS232 Interface

One can combine enumerated types with records to create a very strong, specific definition of an interface. For example, consider the RS232 connection standard. Most RTL's would merely specify it as a 25 bit connection, which could be written in ADLIB as:

```
RS232 = ARRAY [1..251 OF bit;
```

However, this would not make best use of the facilities available. In ADLIB, it would be better to write:

```
TYPE
widerange = (neg_12V,pos_12V);
grounded = (zero);
NETTYPE
RS232 = RECORD
        fg  : grounded;  (*frame ground*)
        td  : widerange;  (*transmit data*)
        rd  : widerange;  (*received data*)
        rts  : widerange;  (* request to send*)
        (* etc.*)
        END;
```

If the net "tty1_line" were declared to be of nettype RS232, then the compiler would accept

```
ASSIGN neg_12V TO tty1_line.td;
```

but would flag as an error:

```
ASSIGN 0 TO tty1_line.terminal_rdy;
```

because of type incompatibility.

CHAPTER 3
A CONTOUR MODEL FOR ADLIB

The chapter is intended to be an informal but unambiguous definition of the behavior of systems specified in ADLIB and SDL, and simulated under SABLE. The definition consists of three parts: a static contour model for ADLIB programs, a dynamic contour model for their execution under SABLE, and a simulation structure that defines the the various ADLIB primitives. For readers unfamiliar with contour models, an introductory tutorial is available in [JBJ71]. Also of interest is SIMULA Begin [BGM73] which is tuned to SIMULA67, and OREGANO [BD71]. However, the basic principles of contour models are fairly simple, and this description will avoid the more complex issues.

3.1 CONTOUR MODELS

A contour model uses rectangles to represent scopes of program identifiers as defined by the user. A static contour model represents the way in which these scopes are nested in the source program, e.g. procedures within procedures, global and local variables etc. Such a drawing can be used to answer questions about identifier visibility, naming conflicts, and data hiding, The set of identifiers visible at any point in the model is determined by examining each enclosing contour in turn. Identifiers inside of non-nested contours can be accessed only through "access pointers" (ap's) that link one scope with another. Access pointers are used extensively in SIMULA 67 to perform a "remote

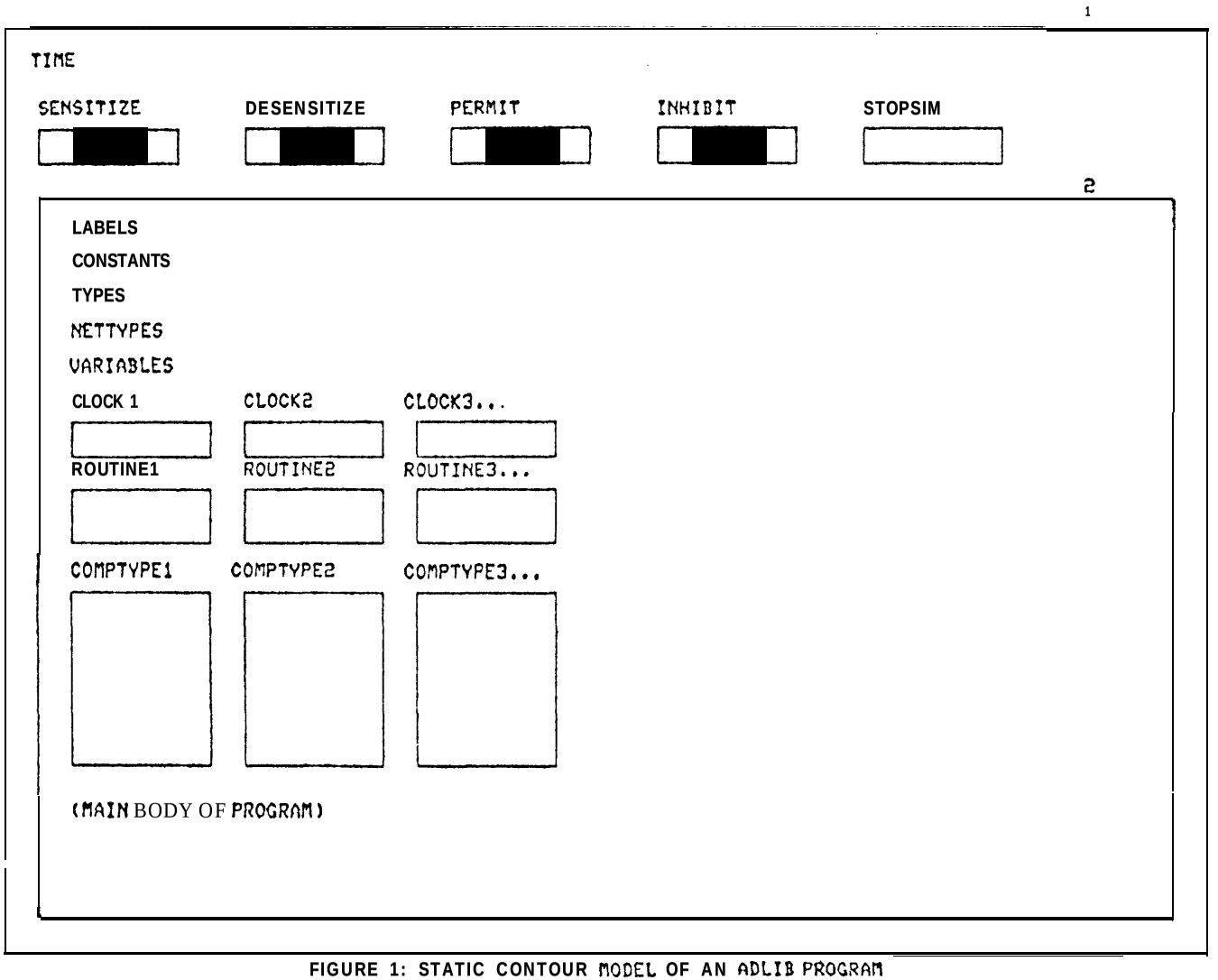


FIGURE 1: STATIC CONTOUR MODEL OF AN ADLIB PROGRAM

access,?' which is where one process reaches into another and may directly alter internal attributes of it. This facility is not available to the ADLIB user directly because it invites hard-to-detect side effects and bad code structuring.

3.2 STRUCTURE OF AN ADLIB PROGRAM

The static contour model for an ADLIB program is illustrated in figure 1.

3.2.1 Global Identifiers

The model shown in figure 1 is not much different from the contour model of a PASCAL program. In the upper left hand corner of each rectangle appear the user defined labels, constants, and types and also the user declared variables*. In addition, more rectangles may appear within a rectangle represent in'g nested scopes.

In the outermost contour is the predeclared variable "time" which represents the simulation time. User assignment to this variable is illegal and is detected at during compilation. Contour 2 in figure 1 is the global level for the user. In it are found the user's global labels, constants, types, nettypes clocks, variables, routines** and comptypes. The meanings of labels, constant and type definitions, are unchanged from PASCAL. There is also an algorithm associated with this

*In this report, items that do not consume storage at runtime are said to be "defined", and items that do are said to be "declared". In particular, items that are defined do not appear inside of contours in the dynamic models.
**Throughout this paper, the word "routine" is taken to mean procedure or function.

contour that could be called the "main body" of the program which can be used for initialization of the global variables, resetting files and such. This main body may not contain any ADLIB control primitives such as detach or waitfor, and may not access any component or net. During the execution of the program's main body, time and all clocks are identically zero. This code block may call global routines that call further routines recursively, just like the main body of an ordinary PASCAL program.

The inclusion of global variables into ADLIB is a concession to practicality and user convenience. Ideally, a design should not have any, since they might represent inter- component connections that have no physical correspondence. However, there are also many applications for global variables that do not violate the intended structure of SABLE. For example? data collect ion and interpretat ion can be simplified if each component calls a routine that accepts intermediate results and stores them away in a global area for later analysis. On the other hand, global variables should not be used for i nt ercomponent communication. This is what nets are intended for, just as parameters are intended for communicating with routines. It may be possible to detect such clandestine component interaction during compilation and prohibit it, just as there have been proposals to ensure that routines have no side effects. However, such mechan i sms can generally be defeated and are invariably unpopular with programmers. The decision of how to use global variables is therefore left to the user.

3.2.2 Nets And Nettypes

Nettype definitions are similiar to type definitions, except that they inform the compiler that it must be prepared to handle nets of this type. Nets are a concept that is unique to ADLIB, similiar to but not the same as ordinary program variables. Nets are allocated in a

different way from variables, and are interconnected with other components and with the simulation support system. The various nettypes thus define the ways that the components are able to interact. The importance of this for error detection and verification is discussed in [HDD79-1]. During simulation, the support system generates new data items of the various nettypes, compares them, does assignments to them, and dynamically regenerates the storage alloted to them. Net types (as opposed to ordinary types) must be used whenever nets are declared inside comptypes or are used as parameters to routines. Functions may not return nets, and nets may not appear on the left side of assignment (":=") statements. To update a net, the ASSIGN statement and the TRANSMIT subprocess facilities are provided. (This is somewhat like SIMULA 67, where updating a pointer requires a special syntax.) Any expression assigned to or compared with a net must be type compatible with the nettype of the net. Within a comptype, two types or net types are compatible if they are subranges of the same base type****.

Whenever a net appears where an expression is called for, the current value of the net is used, in the same way that a program variable normally reflects its value. When a net is used as the subject of an assign or transmit statement, its reference value is used. When a net is declared as a var parameter to a routine (i.e. the keyword var is used in the parameter declaration), the routine may assign to it. If the net is not marked as var, then the routine may access, but not update the value of the net.

**** This is not true in connections between components, because SABLE considers each nettype to be incompatible with all the other nettypes.

3.2.3 Clocks

Following the nettype definitions, an ADLIB program may contain one or more clock definitions. These create functions that map the simulation time to an unsigned integer. Their syntax is:

CLOCK <clockname> (<period> , <numphases>) [DEFAULT1 ;

Symbol ical ly, the function is:

<cl kname> ==
(time/(<period>/<numphases>)) MOD <numphases>

A clock function may be invoked anywhere that a variable integer expression is allowed. Clocks may also be used in timing clauses, which are explained in the next section.

3.2.4 Timing Clauses

Timing clauses may be used in assign, transmit and waitfor statements throughout an ADLIB program. A timing clause yields a time value at each evaluation. This time value is used in various ways as described later. There are two forms of timing clause: sync and delay.

3.2.4.1 SYNC

The syntax for a sync timing clause is:

SYNC [<clock name> [PHASE <integer phase number>]]

If <clock name> is omitted the clock marked “default” is used. If phase is omitted, phase 0 is assumed. The value returned by this sync timing clause is the next time when the specified clock will go through the

specified phase. Symbolically, this can be written as:

SYNC <clock name> PHASE <integer phase number> ==

min ({t' : real | (t' > time) AND
<clockname>(t') = <integer phase number>)

3.2.4.2 DELAY

The other timing expression used in ADLIB is the delay Clause which has the syntax:

DELAY <real delay time>

This evaluates as:

time + <real delay time>

3.2.5 Routines

Global routines are represented as in PASCAL and may include more routines nested within themselves. These routines may be freely called from inside any comptype, and may contain assign statements to net parameters. However, routines may not contain waitfor or detach statements. This restriction allows an enormous simplification and acceleration of the runtime support, because it sharply reduces the need for dynamic storage reclamation ("garbage collection") found in some simulation languages, (for example SIMULA 67). Since ADLIB provides other facilities such as subprocesses and better interprocess communication, it is hoped this restriction will not be overly constraining.

3.2.6 Comptype Definitions

Following the global routine definition section is the "raison d'etre" for the whole ADLIB program, the component type definitions (comptypes). These are the only part of an ADLIB program visible to SABLE. Comptypes are similiar to routine definitions in that they

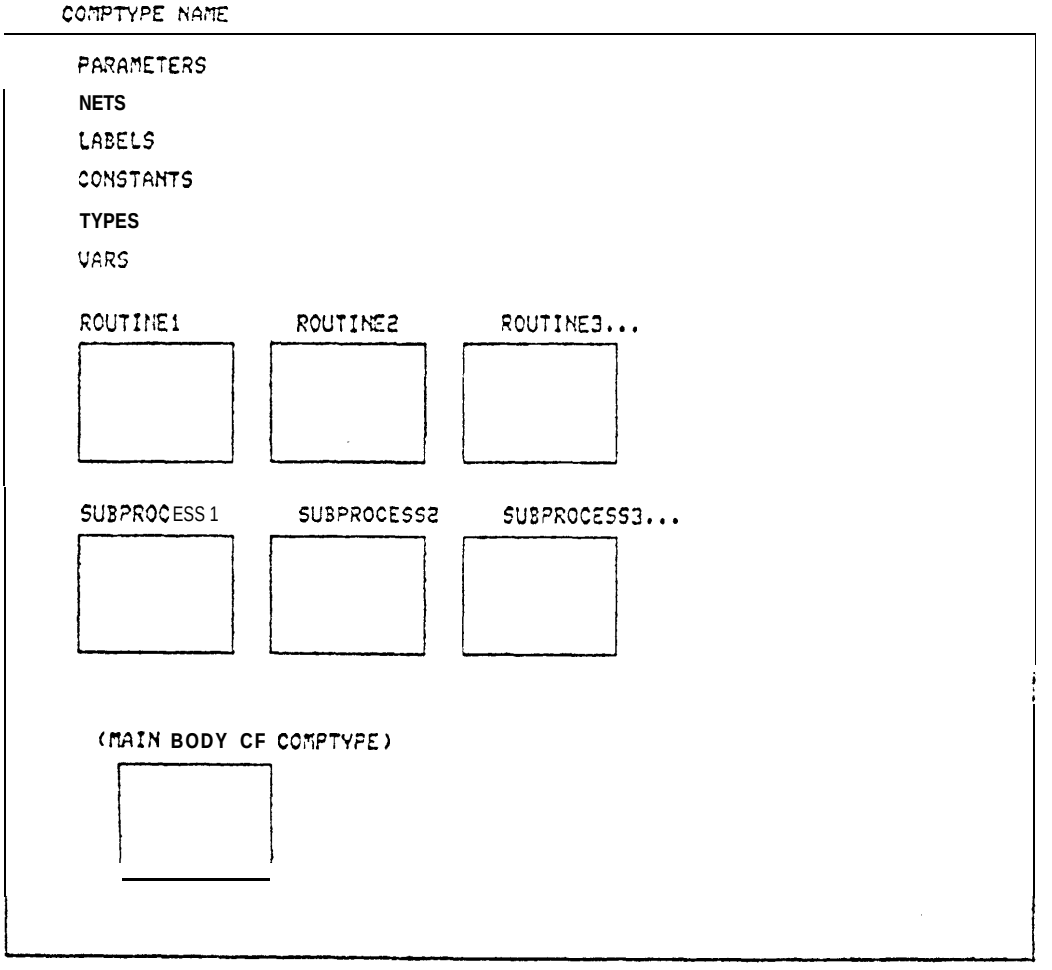


FIGURE 3: STATIC CONTOUR MODEL OF A COMPTYPE

define algorithms for data manipulation. In fact, some functional simulation systems such as BUILD [LT79] use ordinary routine definitions to describe the behavior of components. However, it was felt that this was too inconvienent for the user, and in many cases, the resulting routine was not easily readable. Therefore, ADLIB provides several special features for defining comptypes. The structure of a ccmptype definition is outlined in figure 3, and it includes parameters, defaults, net declarations, labels, constants, types, variables, routines, subprocesses and the "main body" of the comptype.

3.2.6.1 The Heading Of A Comptype

A comptype's parameters are similiar to those of a routine, except that they may be set only in the structure definition language. For example, a comptype "nandgate" might have a parameter "risetime". This would ennble many instances of the nandgate to be allocated by SABLE, with each one potentially having- a different risetime. This is simpler and more efficient than requiring separate models for each. The default section, which follows immediately after the parameter list, may be used to specify default values for any or all of the parameters. The parameters listed in the default section do not have to be in the same order as in the parameter list. All paramet err to comptypes are considered call by value, and pointer, file, and structured data types are prohibited.

3.2.6.2 Net Declarations In A Comptype

Following the parameter default section is the declaration of the nets used by the comptype. These act as the interface between the component and its environment. Nets must be marked as one of the following: INWARD (receive data only), OUTWARD (transmit data only),

EXTERNAL (both receive and transmit) or INTERNAL (receive and transmit but only within the component). It is illegal to assign or transmit to an inward net, or to access the value of an outward net. In addition it is illegal to sensitize an outward net, or to place an outward net in a checklist. The intention here is to ensure that information never flows from a net marked outward or to a net marked inward. If both forms of access are needed the net should be marked external.

3.2.6.3 Internal Nets

INTERNAL nets may be examined and manipulated by a component just like external ones. However, internal nets are part of the behavior specification only, and do not appear in any structural description. They are useful to help specify the behavior of a component with inertial delays or other internal timing characteristics. For example, consider a combinational circuit that implements the function "(a*b)+c" by means of a 2-input AND gate and a 2-input OR gate. If the exact input to output timing relation were important, an internal net might be used to help code this circuit's behavior. In ADLIB, this could be done as in Figure 4.

```
COMPTYPE combin;
INWARD
a,b,c : boolnet;
OUTWARD
d : boolnet;
INTERNAL
x : boolnet;

SUBPROCESS
andgate : TRANSMIT (a AND b) TO x DELAY 15.0;
orgate : TRANSMIT (x OR c) TO d DELAY 14.0;

BEGIN
permit(andgate);
permit(orgate);
END;
```

Figure 4
Combinational Logic: D=(A*B)+C

The internal net x represents the intermediate value (a*b). The transmit subprocess "andgate" causes the value of a*b to be transfered to net x with a delay of 15 time units. The expression "a AND b" is reevaluated whenever net a or net b is updated, and if a new result is obtained a net update takes place. The "orgate" subprocess operates asynchronously so that whenever x or c is updated, an assignment is made to net d. The overall result is that a change in nets a or b is reflected at net d after 29 time units, while a change in net c is reflected after only 14 time units.

Labels, constants, types, variables and routines in comptypes are unchanged from PASCAL, and again these routines may be nested arbitrarily. Following the normal scoping rules, routines have access to all the identifiers inside the comptype, and to all those defined at the global level.

3.2.6.4 Subprocesses

The next part of a comptype definition is the subprocess declaration section. Subprocesses are like "little components" that run autonomously from the main component body, but under its control. They might be used, for example, to describe the direct memory access channels in an IBM 370. Their purpose is to simplify the code in the main body of the comptype by taking care of secondary functions. Because the subprocesses are watching for interrupts and other low level activities, the the main body of a comptype can concentrate on the high level supervisory tasks of the component, resulting in a less cluttered and easier to read piece of code. Subprocesses are less powerful than the main body of the comptype for two reasons: they execute a fixed algorithm to completion each time they are activated, and the criterion for their activation is fixed at compile time, unlike the main body of the comptype which may be stimulated in different ways at different

points in its execution. Subprocesses may be controlled by the predefined procedures

inhibit(<subprocess name>)

and

permit(<subprocess name>)

The former disables a subprocess from running, and the latter enables it to run. These procedure calls may appear anywhere inside a comptype definition, and do not need to textually follow the subprocess named in them.

3.2.6.5 The Main Body Of A Comptype

The main body of the comptype describes the fundamental activities of that type of component. In addition to assigning new values to nets, and permitting and inhibiting subprocesses, the main body may also place itself into a wait state, where it stays until some stimulus is received or some condition is met.

A complete and unambiguous definition of these and all ADLIB primitives in terms of denotational semantics will be available in [HDD79-2].

CHAPTER 4
ADLIB SYNTAX

This summary of ADLIB syntax is derived from the appendices of the PASCAL User's Manual and Report [JK72], with the additional ADLIB constructs included where necessary.

4.1 LOW LEVEL SYNTAX

The basic format of ADLIB programs is patterned closely after PASCAL. However a few points of clarification and difference exist.

1. Identifiers may include the underscore "_", and the use of upper or lower case characters is insignificant.

2. In order to shorten the code, reduce programmer effort, and eliminate transcription errors, a user may "include" files into his or her ADLIB source. The syntax is (starting in column 1):

   #include filename

   "Filename" must be a valid, unambiguous file name. It syntax may depend on the operating system employed.

3. Comments are delimited by "(*" and "*)" and may be nested to any depth. In addition, a second comment convention is supported. Any text between an exclamation point, "!", and the end of a source line is ignored. The exclamation point is ignored inside a (* *) pair, and the symbols (* and *) are ignored between an exclamation point and the end of a line. The following is therefore syntatically

correct:

```
(*this comment is
garbage! notice exclamation is ignored
here ! *)
x := ! but not here, so (* is ineffective
17.2 (* more comments) ;! again is effective
```

and is syntactically equivalent to:

```
x := 17.2;
```

4. In order to pass though the parser, comptype names and net names must be valid ADLIB identifiers. But this can lead to conflicts since designers usually prefer to use names meaningful to their application. For example, one could not define a comptype named "and" or a net named "in" because these conflict with reserved words. To remedy this situation, ADLIB allows a programmer to specify a second name for comptypes, nets and parameters to comptypes. This name is specified immediately after the valid ADLIB name and is enclosed by double quotes. SABLE will see only the name enclosed in quotes. For example:

```
comptype andgate "and"(propdelay "delay" :
            real);
inward
innet "in" : boolnet;
(* etc*)
```

4.2 SUMMARY OF OFERATORS

An asterisk indicates those that are in ADLIB, but not PASCAL.

operator	operat ion	operand	result
ASSIGN*	net assignment	expression,net, timing clause	
:=	assignment	any type except file	
arithmetic:			
+(unary)	ident i ty	integer or real	same as operand
-(unary)	sign inversion		
t	addit ion		
	subtract ion		
*	multiplication		
div	integer division	integer	integer
mod	modulus	integer	
/	real division	integer or real	real
relational :			
=	equality	scalar,string,	boolean
<>	inequality	set or pointer	
<	less than	scalar or string	
>	greater than		
<=	less or equal -or- set inclusion	scalar or string set	
>=	greater or equal -or- set inclusion	scalar or string set	
in	set membership	scalar, and set	
logical:			
not	negation	boolean	boolean
or	disjunct ion		
and	conjunct ion		
set:			
t	union set difference	any set type T	T
*	intersect ion		

4.3 STANDARD IDENTIFIERS

The following are the standard, predefined identifiers in ADLIB. A user is free to use redefine any of them, and implementors are at liberty to include additional predefined constants, types, variables, and routines wherever they might be useful. An asterisk indicates those that are in ADLIB, but not PASCAL.

1. Constants:

   false, true, maxint

2. Types:

   bit*, boolean, char, integer, real, register*, text

3 Files:

   input, output

4 Functions:

   abs, arctan, chr, cos, eof, eoln, exp, ln, odd, ord, pred, round, sin, sqr, sqrt, succ, time*, trunc

5. Procedures:

   desensitize*, detach*, get, inhibit*, new, pack, page, permit*, put, read, readln, reset, rewrite, sensitize*, stopsim*, unpack, write, writeln

4.4 RESERVED WORDS

An asterisk indicates those that are in ADLIB, but not PASCAL. A plus sign indicates DEC10 pascal extension.

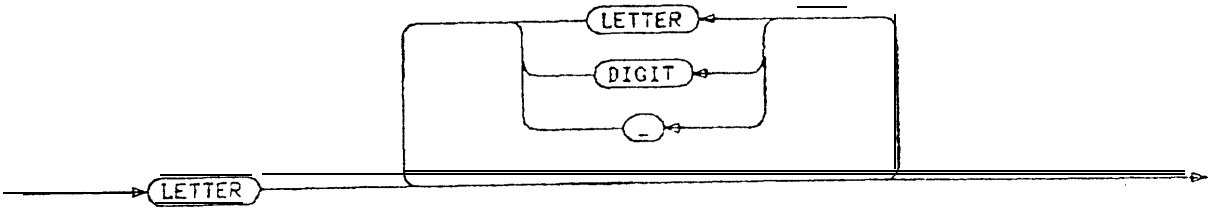
and
array
assign*
beain
case
checks
clock*
comptvpe*
const
default*
delay*
div
do
downto
else
end
ext ern
external*
for
file
fortran+
forward
function
aoto
if
in
inward*
internals
label
mod
nettype*
nil
not
of
others+
out ward*
or
packed
base*
procedure
program
record
repeat
set
subprocess*
sync*
then
to
translator*
transmits
vpe
unt i 1
upon
var
waitfor*
while
with

4.5 SYNTAX CHARTS

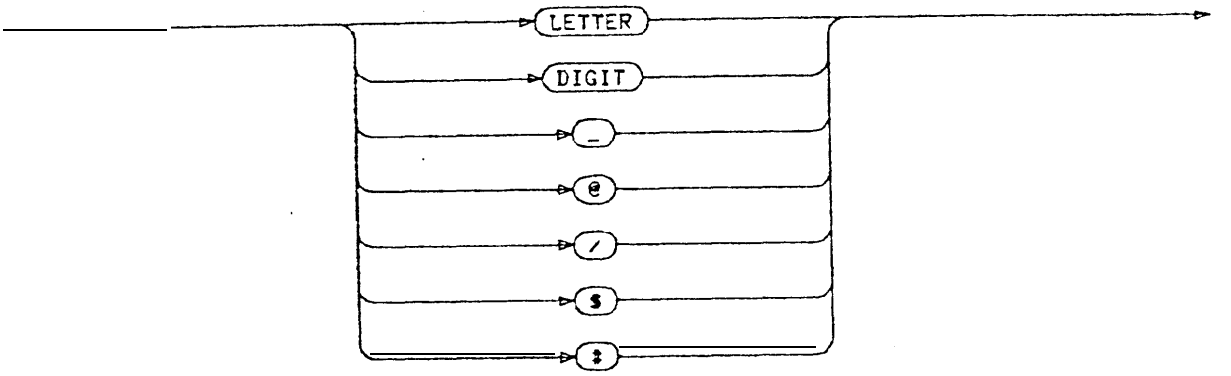
These charts were automatically drawn by a program called Syndia, using a BNF-like notation for input. Documentation on Syndia is available in [CWE79-2].

Field ident, net ident, type ident, subprocess ident, clock ident, nettype ident, constant ident, and variable ident are all syntactically equivalent to ident.

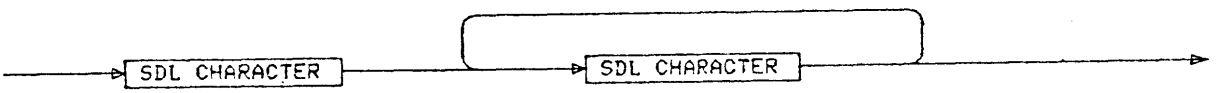
IDENT



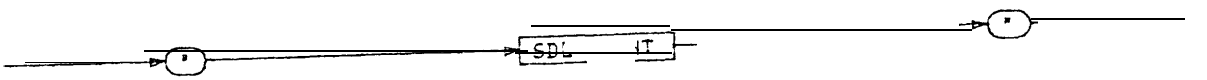
SDL CHARACTER



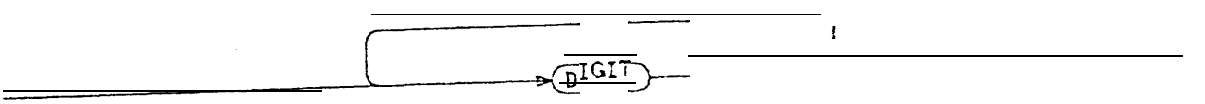
SDL IDENT



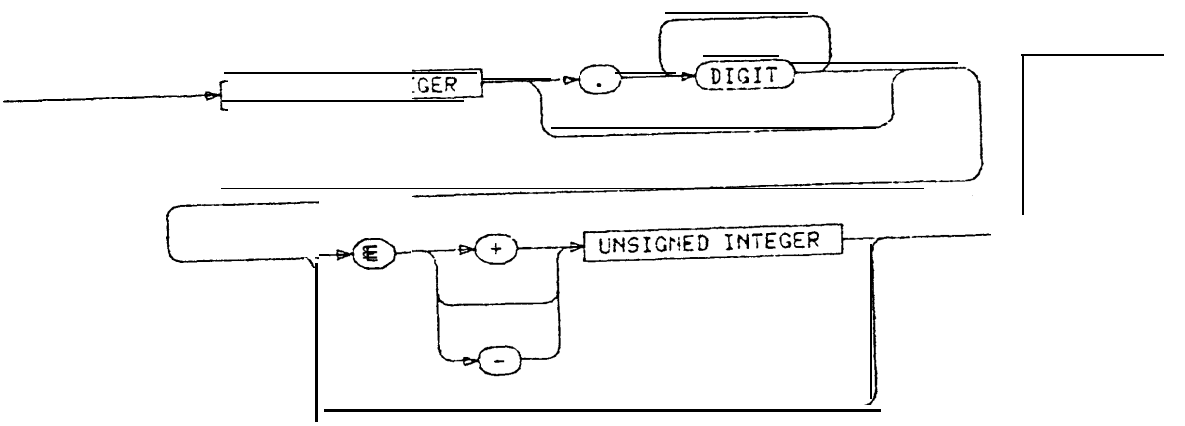
SDL ALIAS



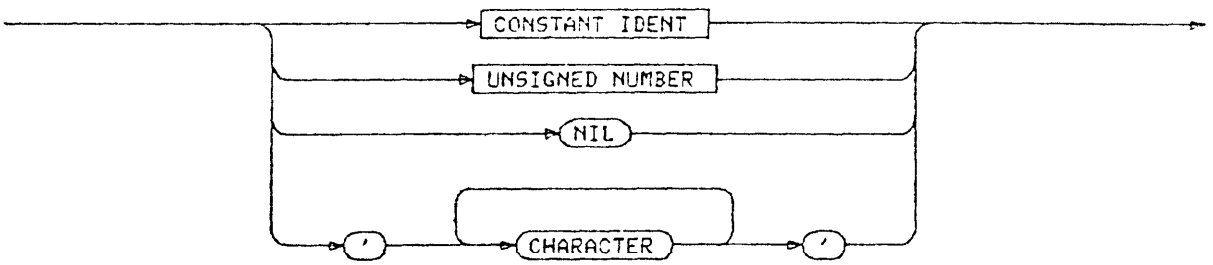
UNSIGNED INTEGER



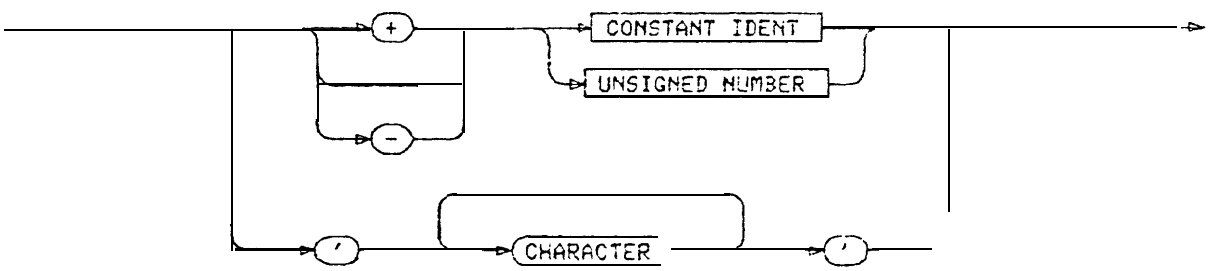
UNSIGNED NUMBER



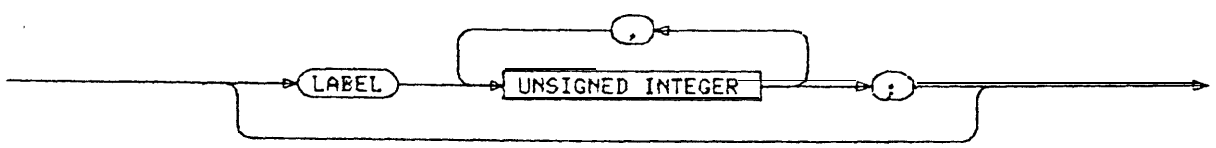
UNSIGNED CONSTANT



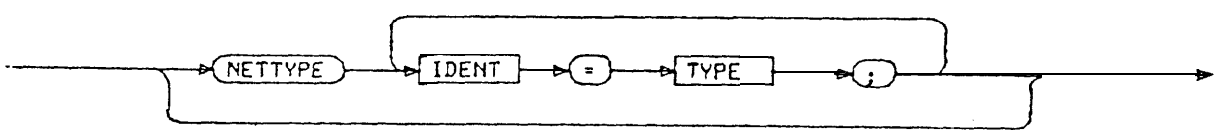
CONSTANT



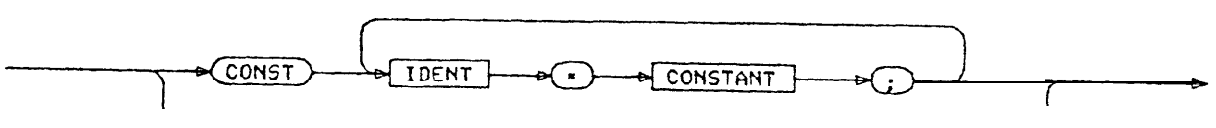
LABEL DEFINITIONS



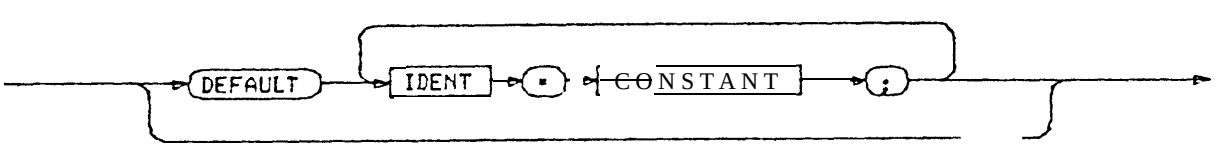
NETTYPE DEFINITIONS



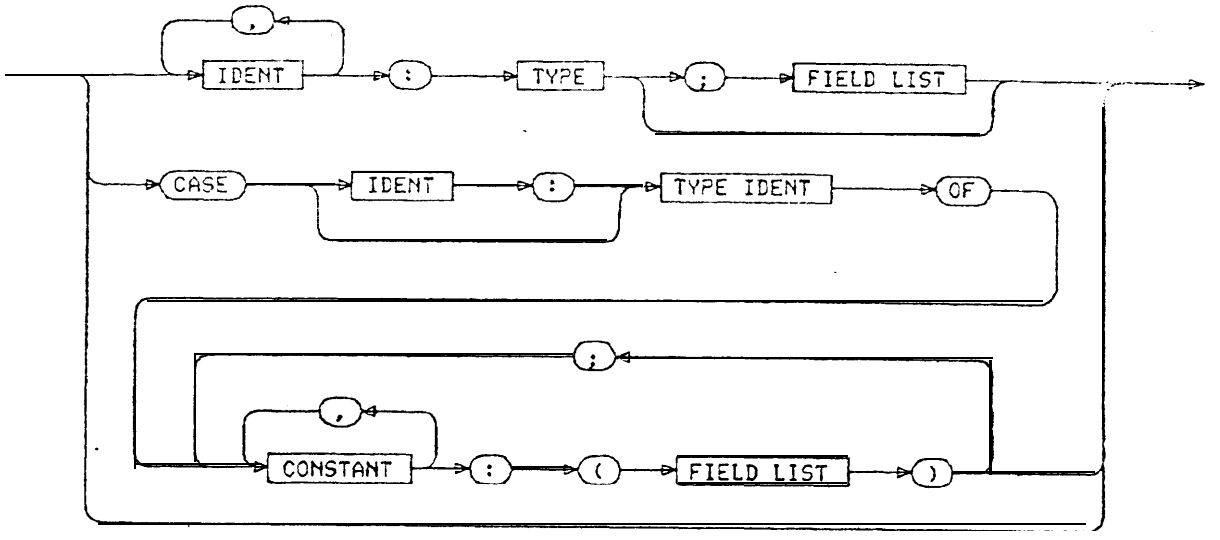
CONSTANT DEFINITIONS



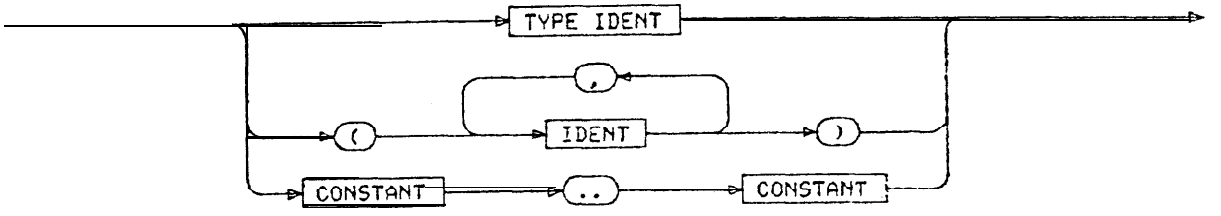
DEFAULTS



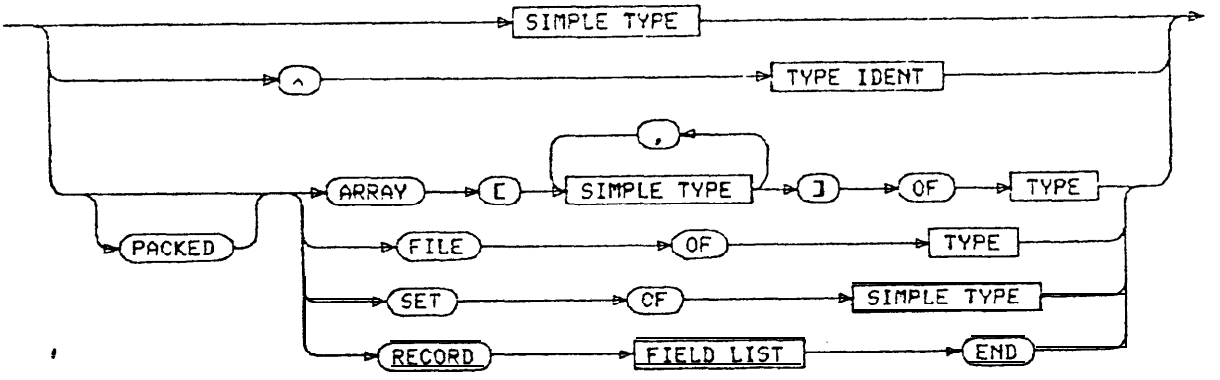
FIELD LIST



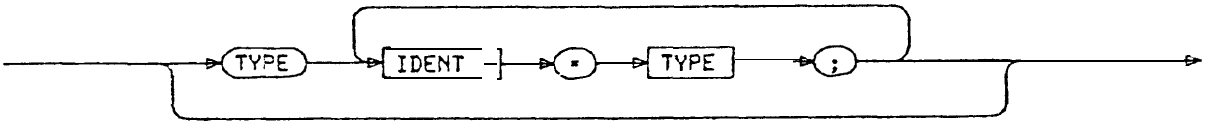
SIMPLE TYPE



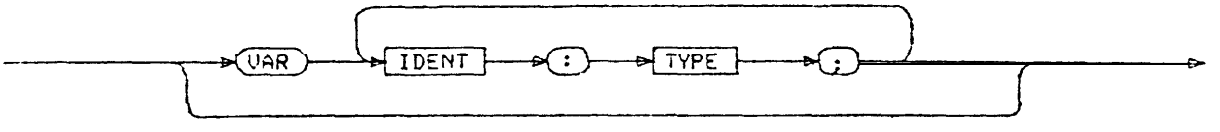
TYPE



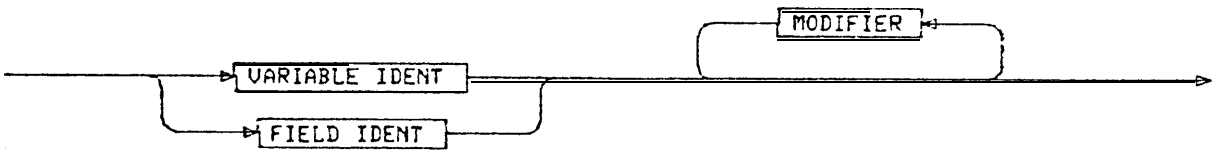
TYPE DEFINITIONS



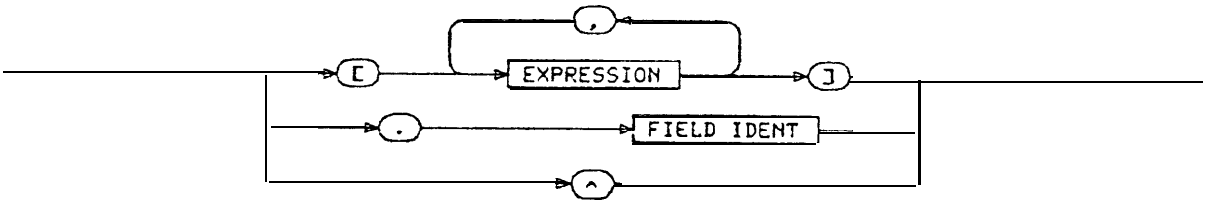
VARIABLE DECLARATIONS



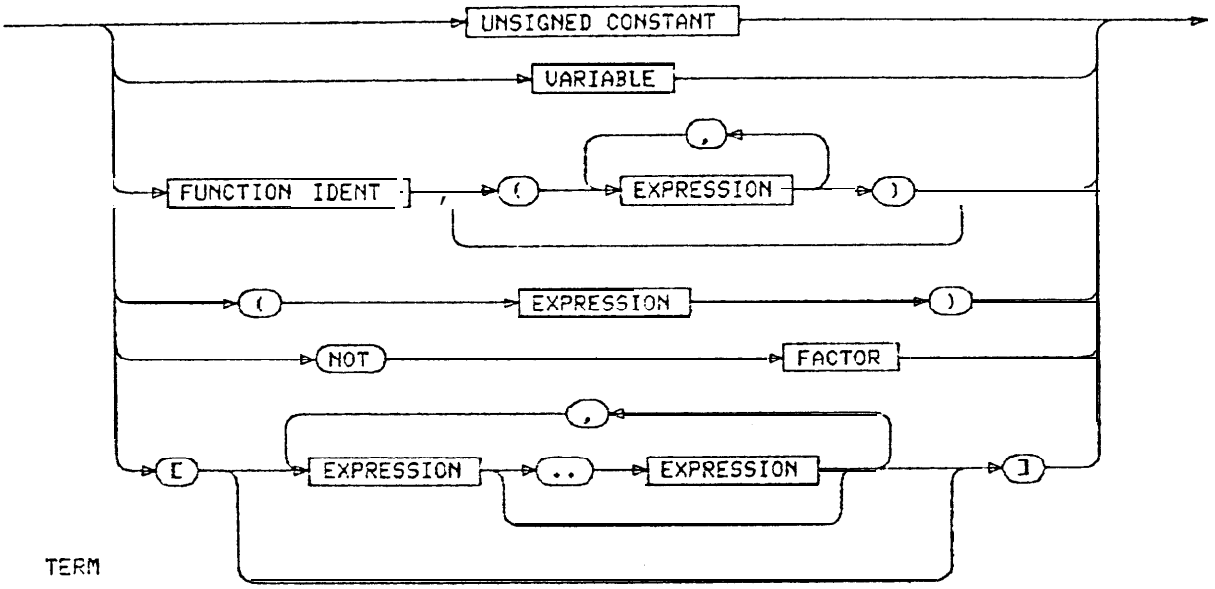
VARIABLE



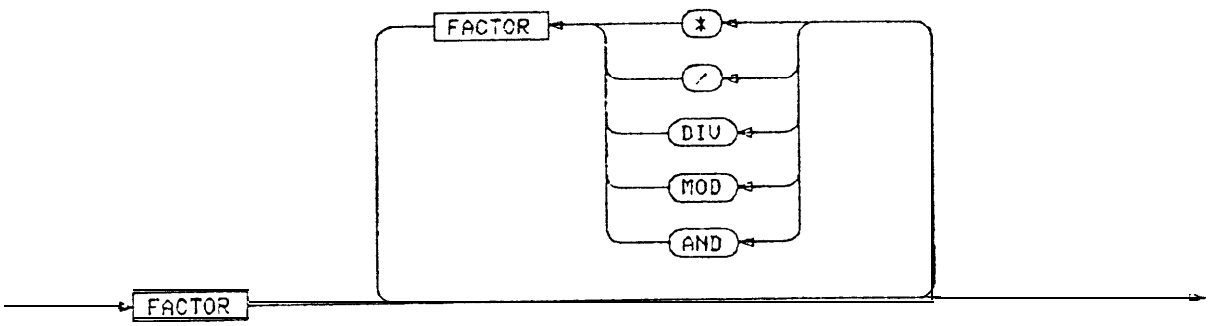
MODIFIER



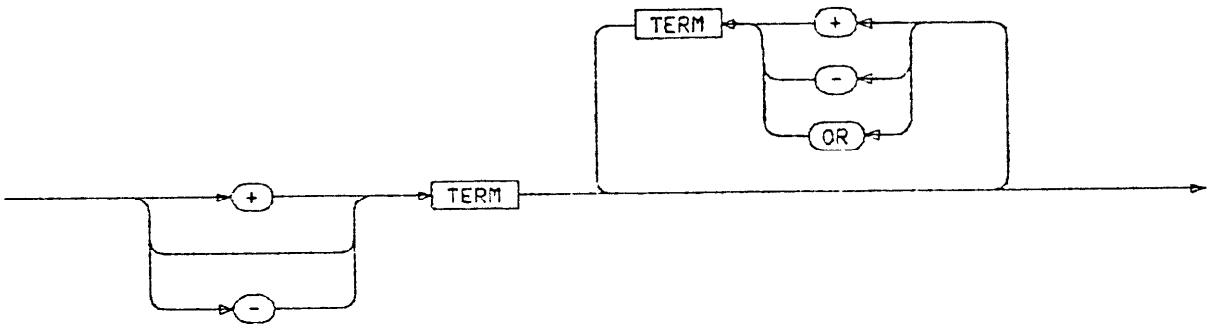
FACTOR



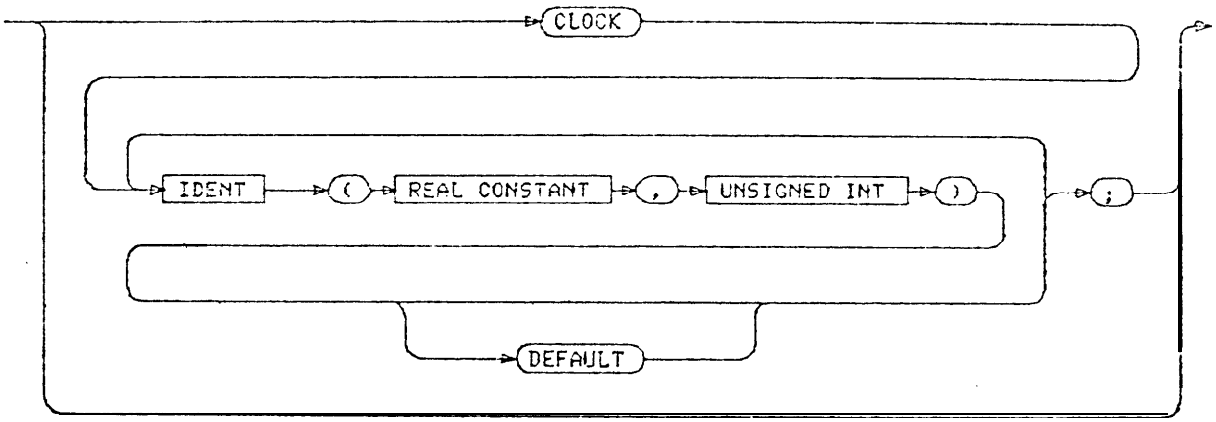
TERM



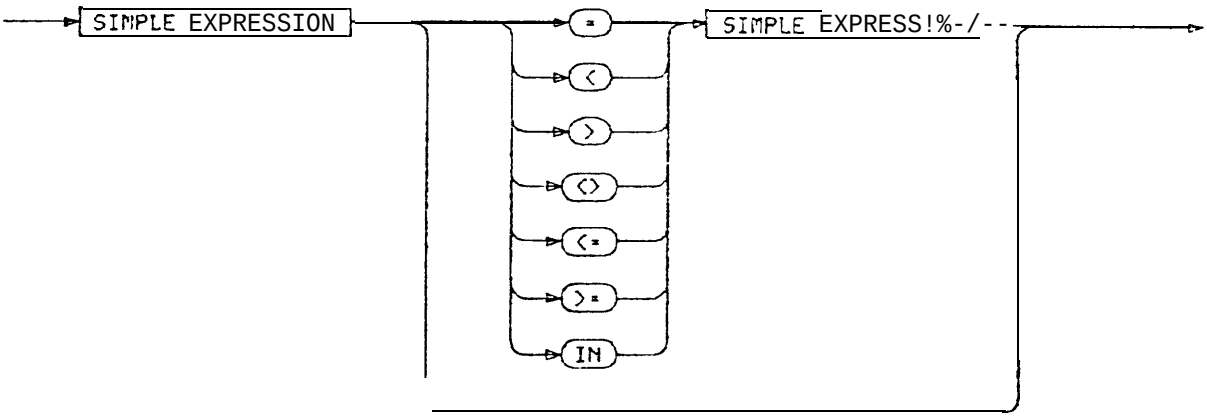
SIMPLE EXPRESSION



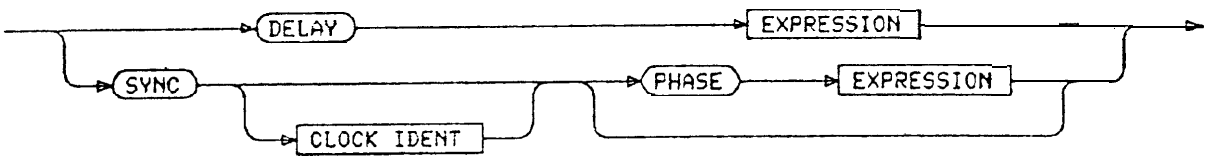
CLOCK DEFINITIONS



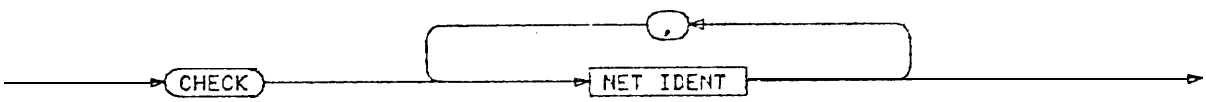
EXPRESSION



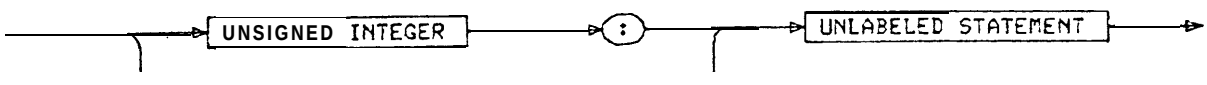
TIMING CLAUSE



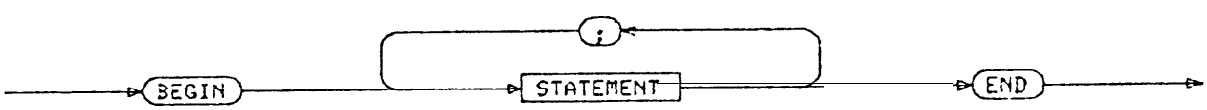
CHECK LIST



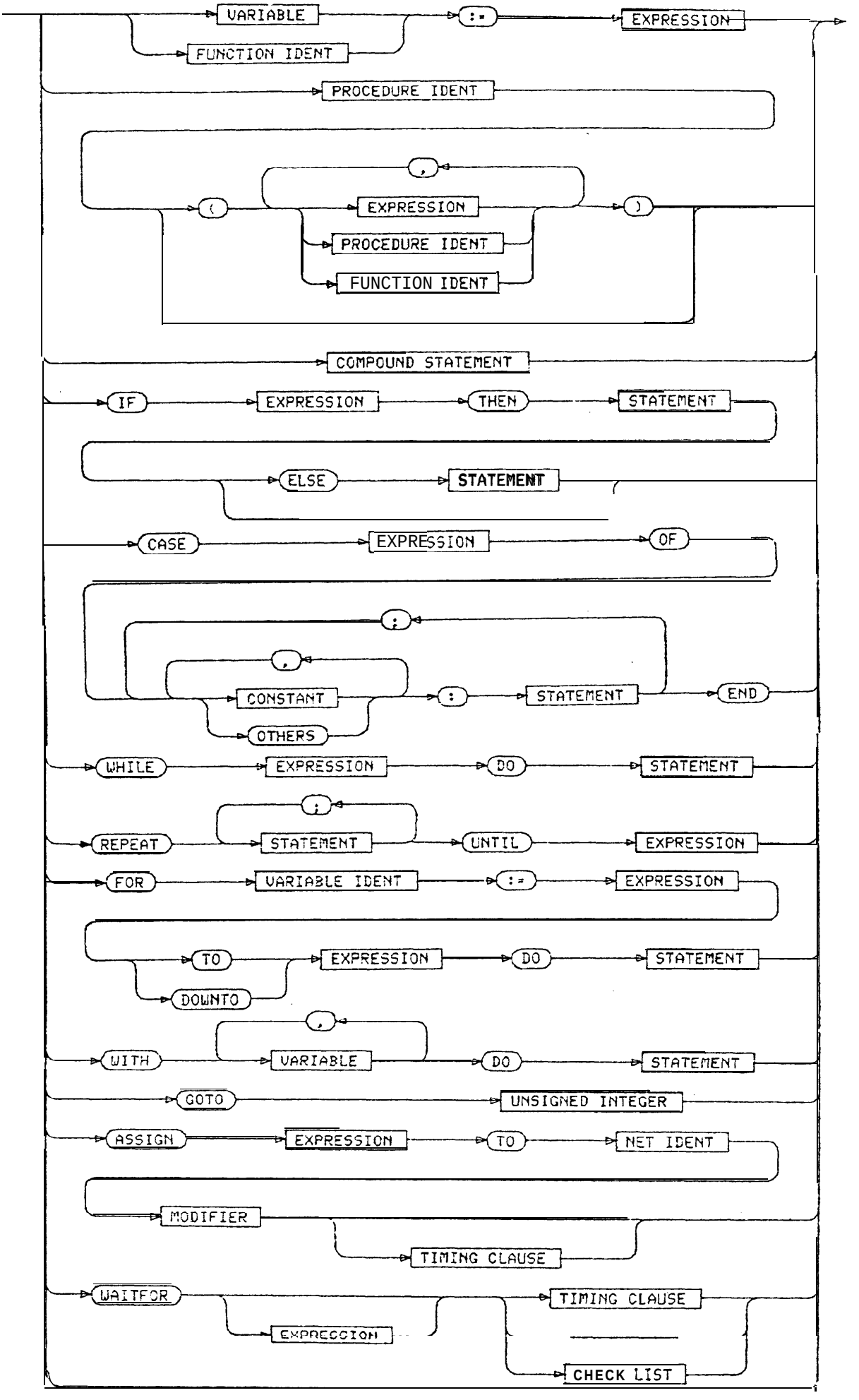
STATEMENT



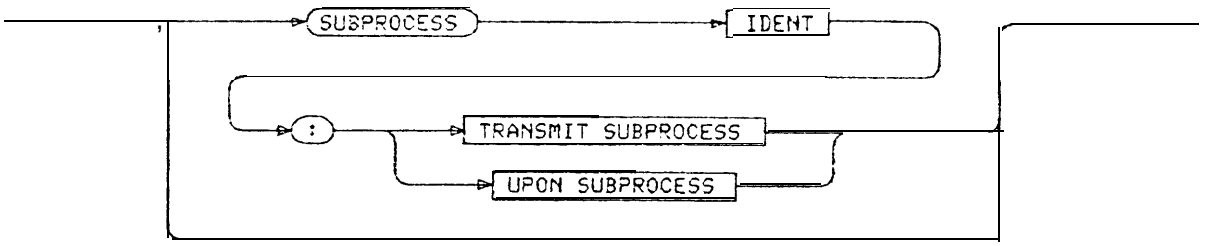
COMPOUND STATEMENT



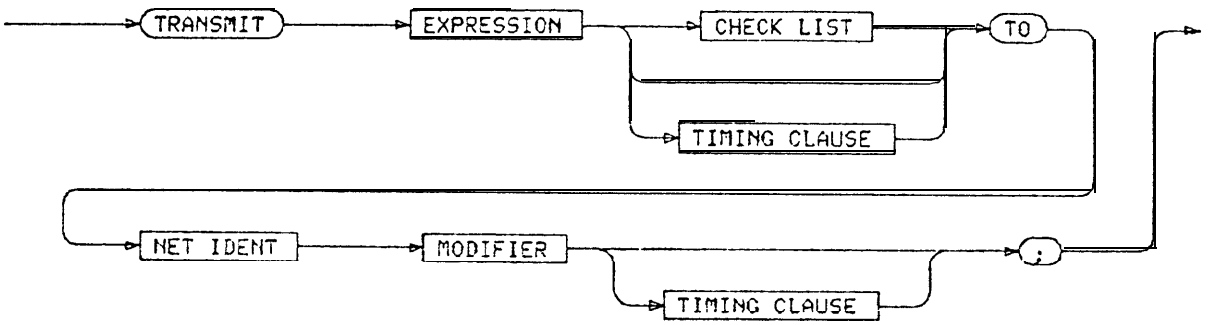
UNLABELED STATEMENT



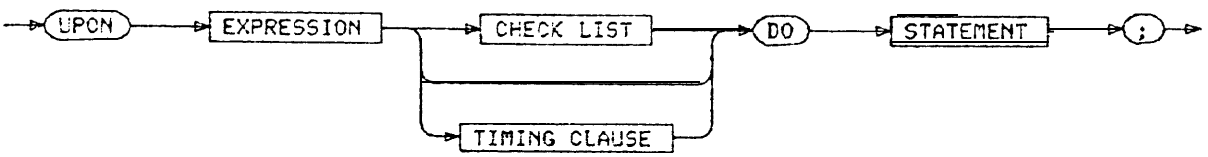
SUBPROCESS **DECLARATIONS**



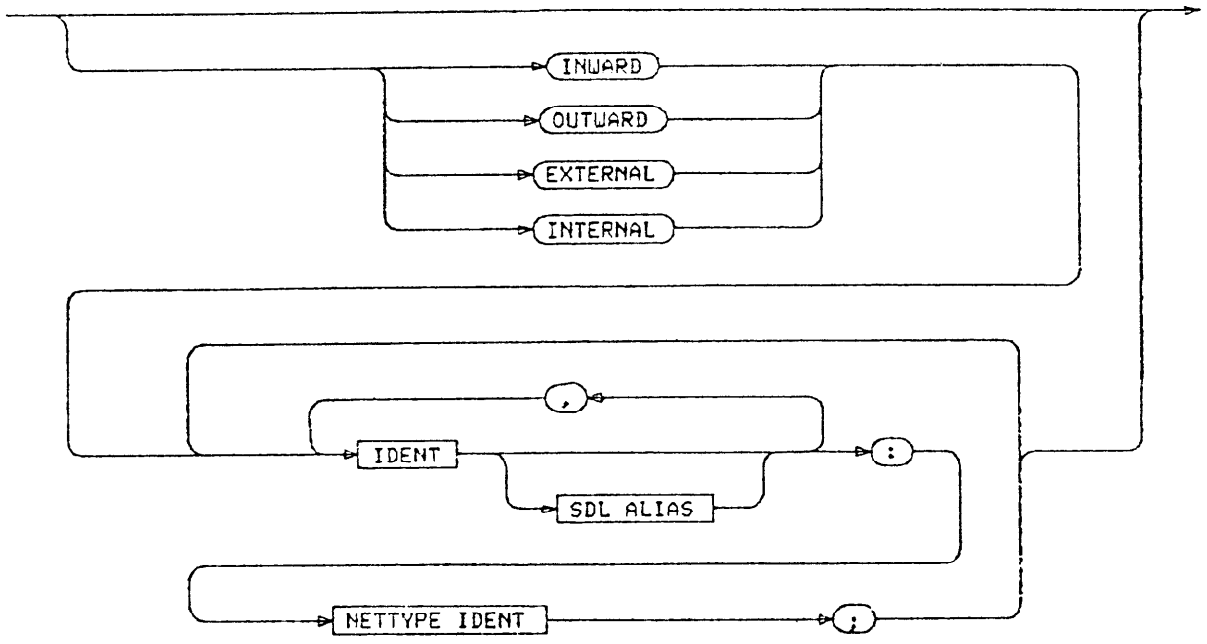
TRANSMIT SUBPROCESS



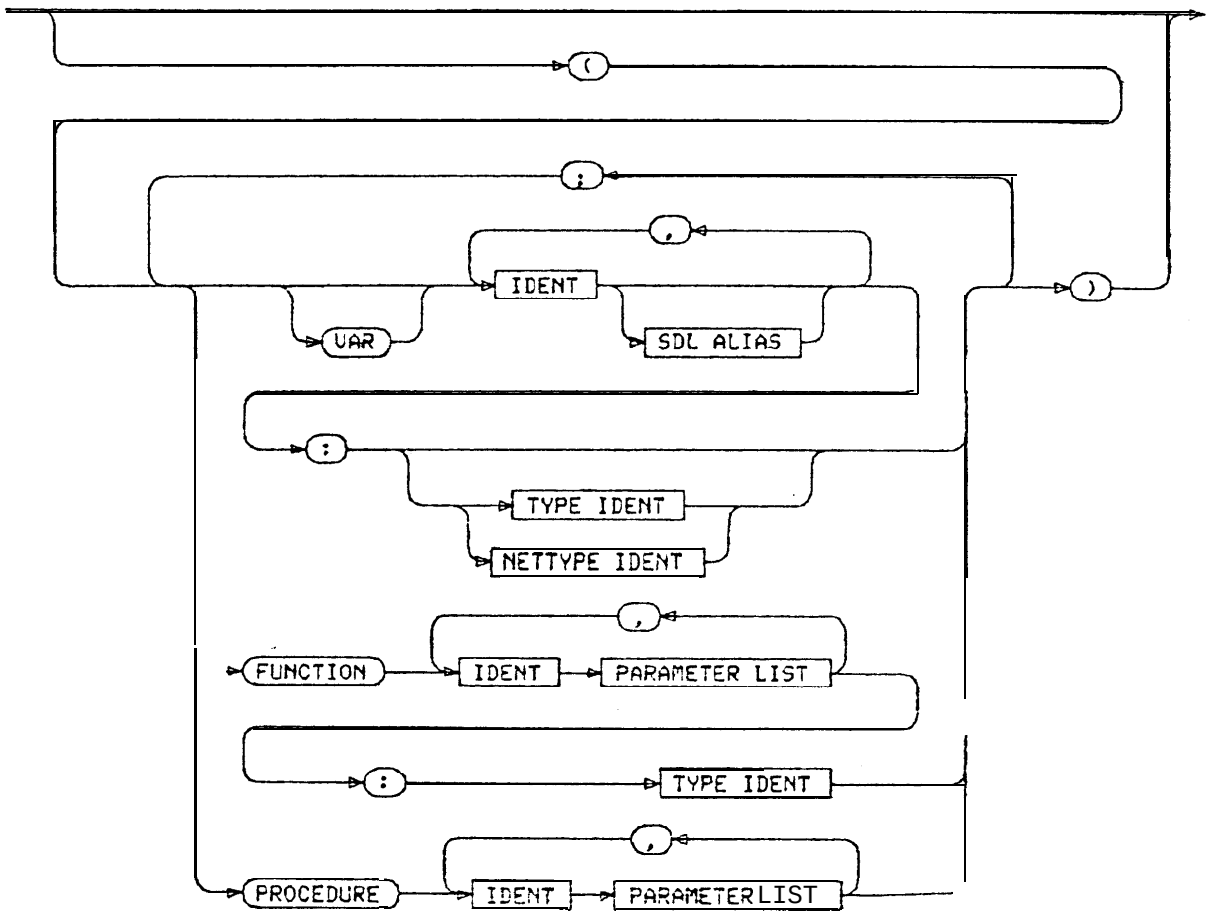
UPON SUBPROCESS



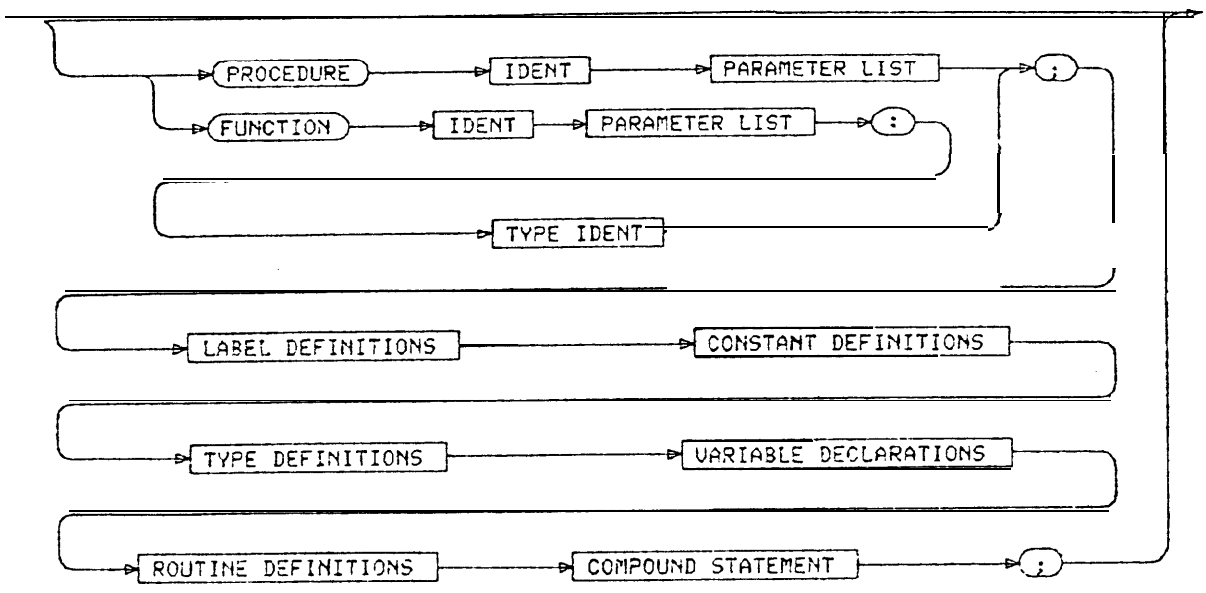
NET REFERENCES



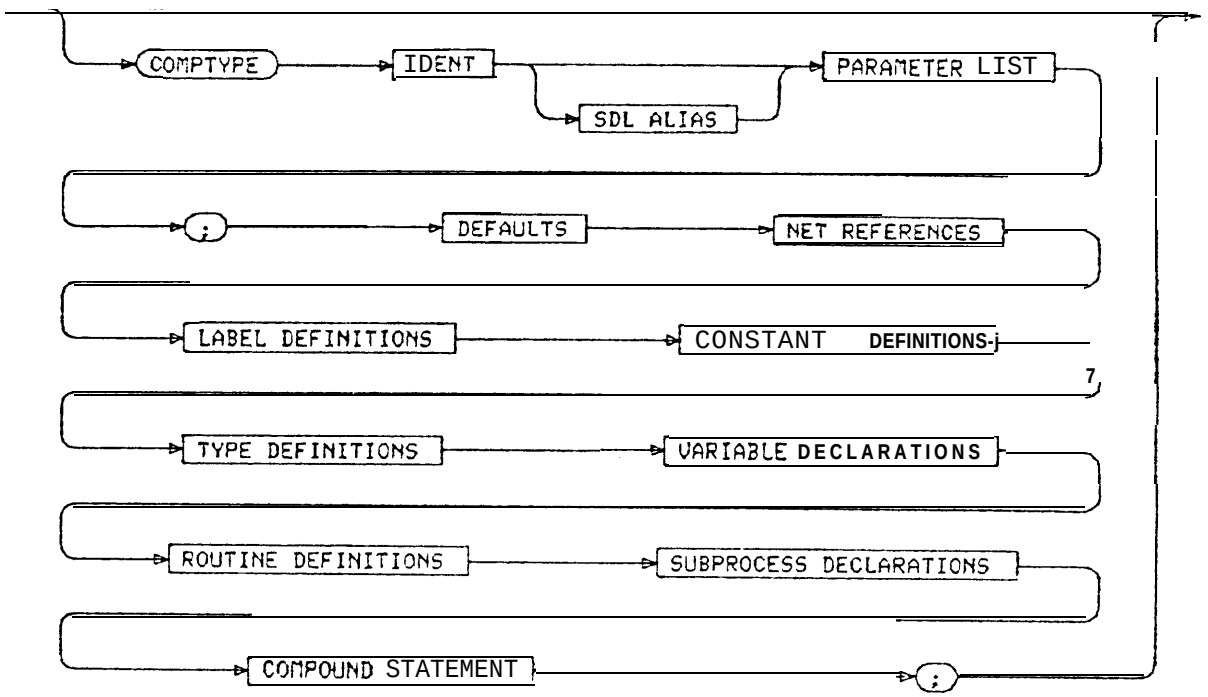
PARAMETER LIST



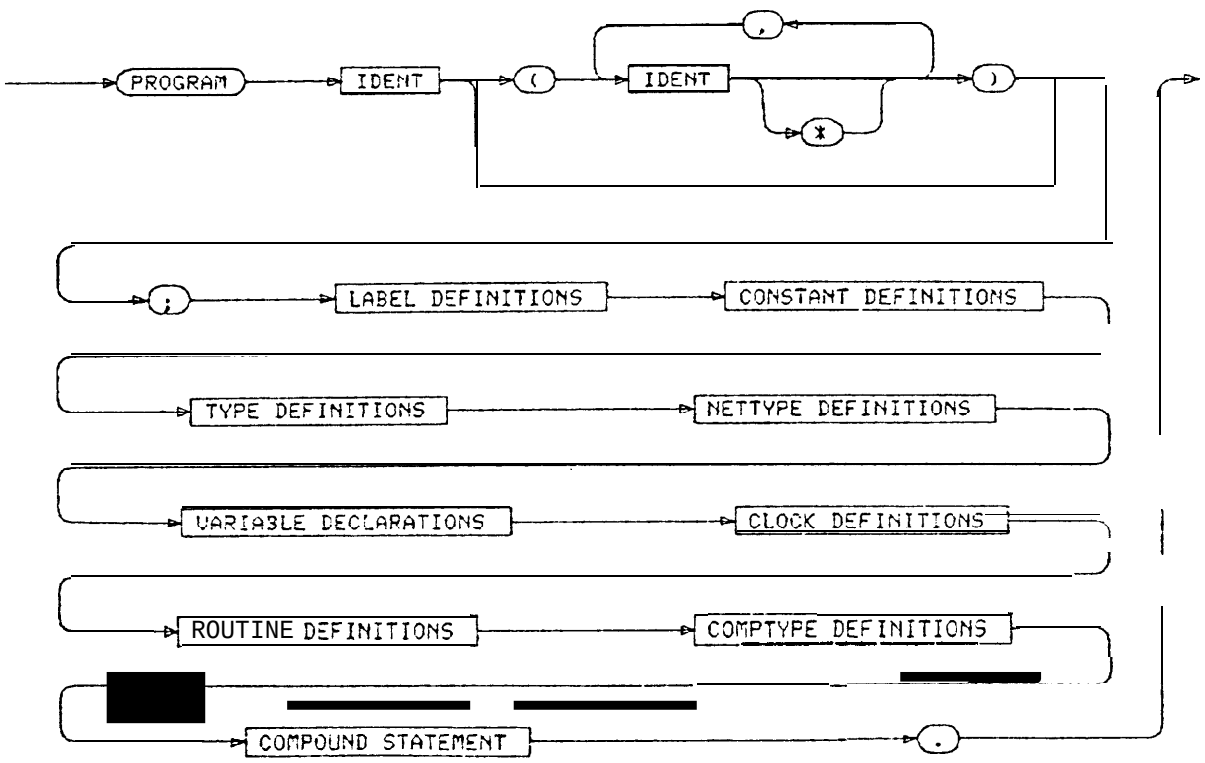
ROUTINE DEFINITIONS



COMPTYPE DEFINITIONS



PROGRAM



APPENDIX A
ROUTINE PACKAGES

A major goal of ADLIB was that the language itself should remain small but be easily extensible by the user or implementor. This is done by adding new data types and new routines. Note that this does not imply any syntax changes or extensions, and does not require the user to learn about features not relevant to his or her own area of work (i.e. a user can not "trip over" an extension that he or she was not aware of.)

In particular, two fairly large packages of routines are available to the ADLIB programmer: Rgpack, which provides bit manipulation facilities not directly available in PASCAL or ADLIB; and Rndpak, which provides a set of random number generators with various distributions. If the implementor did not predeclare them, then the user must declare them in his or her source program in order to access them. The easiest way to do this if with an "include" statement, i.e.:

```
#include rpaks.dcl
```

The file "rpaks.dcl" is assumed to contain the routine headers of all the procedures and functions listed below. Note that all rgpack routines start with the letters "rg" and all rndpak routines start with the letters "rnd". This should help avoid naming conflicts.

A.1 RGPACK

This set of routines is intended to be used with PASCAL and ADLIB programs for describing the bit manipulation features of computers. This has the capability of performing many common hardware manipulations not ordinarily available in higher level languages, such as exclusive or, rotate, etc. They make use of the type "register" and "bit", both of which are type compatible to "integer".

```
Register = integer
bit = 0..1
```

There is no difference between register values and integers for positive values. Negative values are stored in the appropriate format for the negative encoding selected, i.e. one or two's complement, sign magnitude, or unsigned. Registers are stored in one machine word (like integers) and bits to the left of the most significant bit are always 0.

The routines are listed below by class.

A.1.1 Initialization

1. PROCEDURE rgsetup(leastsig, mostsig, format : integer);

Rgsetup must be called prior to calling any other rgpack routine (except the i/o routines). The first two arguments specify the bit number of the least and most significant bits. The third argument specifies the format of negative numbers.

```
two's complement - 0
one's complement - 1
sign magnitude - 2
unsigned - 3
```

For example, to describe an HP21MX computer one would call rgsetup(0,15,0). Unfortunately, the current implementation supports no

more than 34 bits, since it uses ordinary DECSYSTEM-10 PASCAL arithmetic operators, and must avoid machine interrupts on addition.

A.1.2 Conversion

2. FUNCTION rgtoreg(i : integer): register;

Rgtoreg accepts an integer and converts it to register format. An error message will be printed if i is out of range.

3. FUNCTION rgtoint(r : register): integer;

Rgtoint converts a register to integer format.

A.1.3 Arithmetic

The following routines perform most of the common computer arithematic operations in the various formats specified in rgsetup. Two of the routines, rgadd and rgsub also effect bits internal to rgpack itself, so that the user can easily determine if the last addition or subtraction caused an overflow or carry.

4. FUNCTION rgadd(arg1, arg2 : register) : register;

Rgadds adds it two arguments and returns the result, truncated to the number of bits in registers. It sets the overflow if the result is too big for the number of bits, and the carry flag if there was a carry out of the highest position. Note that such a carry does not necessarily imply that the result was too large for the specified word size.

5. FUNCTION rgsub(arg1, arg2 : register1 : register;

Rgsub returns argl - arg2 and sets the overflow and carry flags.

6. FUNCTION rgovrf low: bit;

Rgovrf low returns a one if the overflow flag is set, otherwise zero.

7. FUNCTION rgcarry: bit;

Rgcarry returns a one if the carry flag is set, otherwise zero.

A.1.4 Shifts

8. FUNCTION rgshift(source : register; amount : integer): register;

Rgshift returns its argument shifted the specified number of bit posit ions. (Positive is toward most significant bit.) Padding is with zeros.

9. FUNCTION rgrotate(source : register; amount : integer) : register;

Rgrotate returns its argument rotated the specified number of bit posit ions. (Positive is toward most significant bit.)

10. FUNCTION rgrotlong(source:register; amount:integer; VAR carry:bit):register;

Rgrotlong is similiar to rgrotate except that an extra bit, specified by the user, is included in the rotation.

11. FUNCTION rgarshi ft (source : register; amount : integer): register;

Rgarshift returns its argument shifted the specified number of bit posit ions. (Positive is toward most significant bit.) The arithmetic sign of negative numbers is extended in right shifts.

12. FUNCTION revtranslate(res : integer) : integer;

begin if res<msbpos then res := msbpos - else if res basewordminusl then res := basewordsize;

A. 1.5 Bit Accessing

13. FUNCTION rgbit(r:register; pos : integer) : bit;

Rgbit returns the bit located at position "pos", relative to the least and most significant bits specified in setup. For example, if msbnum = 16 and lsbnum = 1 then rgbit(6,3) will return the third bit from the right (lsb) side, which is 1, since 6 is represented as 0000000000000110 in binary.

14. FUNCTION rgbitset(r:register; pos : integer; newval : bit) : register;

Rgbit sets the bit located at position "pos", relative to the least and most significant bits specified in setup to "newval" and returns the result.

15. FUNCTION rgfield(r:register; left 'right : integer) : register;

Rgfield returns the bits located between position "1eft" and "right", relative to the least and most significant bits specified in setup. For example, if msbnum = 16 and lsbnum = 1 then rgfield(6,4,2) will return 3 (011 in binary.>

16. FUNCTION rgfldset(r:register; left 'right : integer; newval : register):register;

Rgfldset sets the bits located between and including posit ions "left" and "right", to "newval", and returns the results.

17. FUNCTION rgfdr1(rg : register) : integer;

Rgfdrl finds the rightmost 1 in a register, and returns its bit position in the coordinates used at setup time. If no one is found, the bit number of the bit to the left of the most significant bit is returned.

18. FUNCTION rgfdr0(rg : register) : integer;

Rgfdr0 finds the righmost 0 in a register, and returns its bit position in the coordinates used at setup time. If no one is found, the bit number of the bit to the left of the most significant bit is returned.

19. FUNCTION rgfdl1(rg : register) : integer;

Rgfdll finds the leftmost 1 in a register, and returns its bit position in the coordinates used at setup time. If no one is found, the bit number of the bit to the right of the least significant bit is returned.

20. FUNCTION rgfdl0(rg : register) : integer;

Rgfdl0 finds the leftmost 0 in a register, and returns its bit posit ion in the coordinates used at setup time. If no zero is found, the bit number of the bit to the right of the least significant bit is returned.

A.1.6, Logical

21. FUNCTION rgand(arg1,arg2 : register) : register;

Rgand returns the bitwise AND of its arguments-

22. FUNCTION rgnand(arg1,arg2 : register) : register;

Rgnand returns the bitwise NAND of its arguments.

23. FUNCTION rginv(arg1: register) : register;

Rginv returns the bitwise INVERSION of its argument.

24. FUNCTION rgor(arg1,arg2 : register> : register;

Rgor returns the bitwise OR of it s argument s.

25. FUNCTION rgnor(arg1,arg2 : register) : register;

Rgnor returns the bitwise NOR of its arguments.

26. FUNCTION rgxor(arg1,arg2 : register> : register;

Rgxor returns the bitwise EXCLUSIVE OR of its arguments.

27. FUNCTION rgnxor(arg1,arg2 : register> : register;

Rgnxor returns the bitwise inversion of the EXCLUSIVE OR of its arguments, (which is also called the equivalence relation.)

A. 1.7 Formatted I/O

28. FUNCTION rgrdoct(VAR f : text) : integer;

Rgrdoct reads a [signed] octal number from the specified file and returns it as an integer. This can be assigned directly to a variable of type register, if desired.

29. FUNCTION rgrdhex(VAR f : text) : integer;

Rgrdoct reads an [signed] hexadecimal number from the specified file and returns it as an integer. This can be assigned directly to a variable of type register, if desired.

30. FUNCTION rgrdbin(VAR f : text) : integer;

Rgrdoct reads an [signed! binary number from the specified file and returns it as an integer. This can be assigned directly to a variable of type register, if desired.

31. PROCEDURE rgwtoct(VAR f:text; i,width : integer);

Rgwtoct writes i into file f as an octal number padded on the left with blanks so that "width" characters are printed. Max width=40.

32. PROCEDURE rgwtbin(VAR f:text; i,width : integer);

Rgwtbin writes i into file f as a binary number padded on the left with blanks so that "width* characters are printed. Max width=40.

33. PROCEDURE rgwthex(VAR f:text ; i,width : integer>;

Rgwthex writes i into file f as a hexadecimal number padded on the left with blanks so that "width" characters are printed. Max width=40.

A.2 RNDPACK

Rndpack is a set of fairly random number generators useful for stochastic simulations. They are 1 isted below.

A.2.1 Setting A New "Seed"

1. PROCEDURE rndset(newseed : integer);

Rndset resets the internal random number generator mechanism according using "newseed". It is only necessary to call rndret if multiple simulation runs are to be performed using different "random" inputs.

A.2.2 Random Drawing Functions

2. FUNCTION rnd01 : real ;

Rnd01 produces a random number between 0 .O and 1 .O by using one random number generator to scramble the results of another, thereby lowering the autocorrelation.

3. FUNCTION rndnexp(lambda : real> : real ;

Rndnexp returns a number drawn from the negat i ve exponent ial distribution with mean and standard deviation 1.0/lambda (lamba must be positive>.

4. FUNCTION rnderlang(lambda : real; k : integer> : real ;

Rndelrang returns a number drawn from the Erlang distribution with mean (1/lamba) and standard deviation 1/(sqrt(k)*lambda). (Minimum k=1, higher k makes for a tighter distribution.)

5. FUNCTION rndnormal(mean, variance : real) : real ;

Rndnormal returns a number drawn from the Normal distribution with the mean and variance specified. The distribution is approximated by summing 36 uniformly distributed random values.

6. FUNCTION rndint(low, high : integer) : integer;

Rndint produces an integer evenly distributed among the numbers from low to and including high.

7. FUNCTION rnddraw(p : real) : boolean;

Rnddraw returns true with probability p.

A.2.3 Data Analysis Facility

8. FUNCTION rnduniform(low, high : real) : real ;

Rnduniform produces a random real number uniformly distributed between low and high. (Note that the probability of returning a value exactly equal to low or high is vanishingly small.)

9. PROCEDURE rndhisto(data : real ; command : integer);

Rndhisto collects, analyzes and plots a random variable. It produces a histogram automatically scaled to the width of the paper. Maximum number of bins= 200. Its "commands" are as follows:

```
0 reset all tallies, and data values
1 set high limit to "data"(default = 10.0)
2 set low limit to "data"(default = -10.0)
3 set number of bins to "data"(default = 20)
4 accept "data" as a point to be plotted
5 plot results in file **output**
6 plot results at tty
7 set paper width to "data"(default = 79 columns)
8 reset all parameters to default values
```

The print out includes: the number of points, the value of the highest and lowest points, the number of points out of range high and low (if any), the mean, variance, sum, standard deviation, sum of squares' and the auto covariance and autocorrelation of adjacent terms. The following trivial program shows an example of its use:

```
program x ;
var i : integer;
function rnderlang(lambda : real; k : integer> : real; exfern;
procedure rndhisto(data : real; command : integer);extern;
begin
for i := 1 to 1000 do rndhisto(rnderlang(3.0,4),4);
rndhisto(0.0,5);
end.
```

This produces the output shown:

```
-9.50000              0>
-8.50000              0>
-7.50000              o>
-6.50000              o>
-5.50000              o>
-4.50000              o>
-3.50000              o>
-2.50000              o>
-1.50000              o>
-5.00000E-01          7>
 5.00000E-01        134>XXXXXXXXXXXXXX
 1.50000            297>XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 2.50000            262>XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 3.50000            154>XXXXXXxxxxxxxxxxx
 4.50000             82>XXXXXXXXX
 5.50000             32>XXX
 6.50000             17>X
 7.50000              8>
 8.50000              6>
 9.50000              0>
 NUM POINTS=          1000 LOWVAL= 1.826466932E-01
HIGHVALUE= 1.178417652E+01
 1 POINT(S) WERE TOO HIGH   0 POINT(S) WERE TOO LOW
 MEAN= 2.976368099       VARIANCE= 2.365632474
 SUM= 2.976368099E+03
 SUMSQ= 1.122203394E+04 SUM PROD= 8.854801416E+03
 STD DEV= 1.538061276
 AUTO COVARIANCE=-6.792426109E-03
 AUTOCORRELATION=-2.871293902E-03
```

The code for this routine is not very complicated, so the user may wish to copy it and modify it for his or her own special application.

APPENDIX B

REFERENCES

In addition to those references specifically sited in this manual, some additional works in the areas of language, simulation, and design automation are included here for the interested reader.

[AM77] Abramovici, M., M.A. Breuer, K. Kumar, "Concurrent Fault Simulation and Functional Level Modeling," Proc. 14th Design Automation Conference, New Orleans, 1977, pp. 128-137.

[BD71] Berry, D., 'Introduction to Oregano," Siaplan Notices, Feb. 1971, pp. 171.

[BCG71] Bell, C. G., A. Newell, Computer Structures: Readings and Examples, McGraw Hill, New York, 1971, pp. 22-33.

[BGM73] Birtwistle, G. M., O-J Dahl, B. Myhrhaug, K. Nygaard, SIMULA BEGIN, Auerbach Publishers, Inc., Philadelphia, Pa. 1973.

[CSG76] Chappell, S. G., P. R Menon, J. F. Pellegrin, A. Schowe. '*Functional Simulation in the Lamp System," Proceedinqr of the 13th Design Automation Conference, San Francisco, 1976, pp 42-47.

[CY74] Chu, Yaohan, '*Introducing CDL," Computer, December, 1974.

[CHD74] Coplaner, H. D., and J. A. Janhu, "Top Down Approach to LSI System Design." Computer Design Vol. 13, No. 8, Aug 1974, pp.

143-148.

[CWE79] Cory, W. E., J. R. Duley, W. M. vanCleemput, An Introduct ion to the DDL-P Language, Computer Systems Lab. Technical Report No. 163, Stanford University, Stanford, Ca. March 1979.

[CWE79-2] Cory, W. E., Syndia User's Guide Systems Lab. Technical Report No. 176, Stanford University, Stanford, Ca. August 1979.

[DOJ70] Dahl, 0. J., B. Myhrhaug and K. Nygaard, S imula Common Base Lanauaae, Norwegian Computing Center, Oslo, Publicat ion s-22, Oct. 1970.

[DDL75] Dietmeyer, D. L., and J. R. Duley, "Regist er Transfer Language5 and Their Translation," in Digitai System Design Automat ion, edited by M. A Breuer, Computer Science Press, inc., Woodland Hills, Ca., 1975, pp 117-218.

[FD78] Ferrari, Domenico, Computer Svst ems Performance Evaluation, Prentice- Hill, Inc. Englewood Cliffs, New Jersey, 1978.

[GRI77] Gardner, Robert I., "Multi-Level Modeling in SARA," Proc. of Symposium on Desisn Automat ion and Microprocessors, Palo Alto, Feb. 24-25, 1977, pp. 63-67.

[HP] Hewlett-Packard, A Pocket Guide to Hewlett-Packard Computers, Palo Alto, Ca.

[HPB77] Hansen, Per Brinch, The Architecture of Concurrent Programs, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[HDD79-1] Hill, Dwight D. and W. M. vanCleemput, "SABLE : A Tool for Generating Structured, Multi-level Simulations", Proceedings of the 1979 Design Automation Conference, San Diego, Ca.

[HDD79-2] Hill, D.D., Multi-Level Simulator for Computer Aided Desian, Ph. D. Thesis, Center For Integrated Systems, Stanford U., 1979.

[HFJ73] Hill, Frederick J. and Gerald R. Peterson, Digital Svstem Hardware Organization and Desian, John Wiley and Sons, 1973.

[I73] Intel, 8008 8 Bit Parallel CPU Users Manual, Santa Clara, C-3, 1973.

[JK74] Jensen, K. and N. Wirth, Pascal User Manual and Report, Springer-Verlag, New York, 1974.

[JBJ71] Johnston, John B., "The Contour Model of Block Structured Processes," Sigplan Notices, Feb. 1971, pp. 55.

[KPJ69] Kiviat, P.J., R. Villanueva, and H. Markowitz, The SIMSCRIPT II Programming Language, Prentice Hall, Inc., Englecrood Cliffs, N.J. 1969.

[KZ70] Kohavi, Zvi, Switching and Finite Automata Theory, McGraw- Hill Book Company, San Francisco, Ca., 1970.

[LT79] Langlet, T., private communication, Burroughs Corporation, Mission Viejo, Ca. 1979.

[LP75] Losleben, P., Design Validation in Hierarchical Systems, Proc 12th Design Automation Conference, Boston, 1975, pp. 431-438.

[MMH75] MacDougall, M.H., "System Level Simulation," in Digital Svrtem Desian Automation, edited by M. A Breuer, Computer Science Press, Inc., Woodland Hills, 1975, pp. 35-62.

[MB77] Magnhagen, Bengt, Probabilitv Based Verification of Time Margins in Digital Designs, Department of Electrical Engineering, Linkoping University, Linkoping Sweden.

[MWTM78] McWilliams, T.M. and L. C. Widdoes, SCALD: Structured Computer-Aided Logic Design, Digital Systems Lab. Tech Report No. 152, Stanford U., March 1978.

[NRA781 Newton, R. A., The Simulation of LSI Circuits, University of Cal., Berkeley, Memo No. UCB/ERL M78/52, July 1978.

[OEI78] Organick, Elliott I., A.I. Forsythe, R.P. Plummet-, Programming Language Structures, Academic Press, San Francisco. 1978.

[RJ77] Rohmer, J. "SSH Simulateur de Systems Hierarchises," Proc. 10th Annual Simulation Symposium, 1977, pp. 109-127. Inc., Woodland Hills, 1975, pp. 117-218.

[RCW75] Rose, C.W., and M. Albarran, "Modeling and Design Description of Hierarchical Hardware / Software Systems," Proc. 12th Desian Automation Conference, Boston, 1975, pp. 421-430.

[TM78] Tokoro, M., M. Sato, M. Ishigami, E. Tamura, T. Ishimitsu, H. Ohara, A Module Level Simulation Technique for Svstems Composed of LSI's and MSI's, Proc. 15th Design Automation Conference, Las Vegas, 1978, pp. 418-427.

[US79] Utt, Steve, SUDS2 User's Guide, Department of Electrical Engineering, Stanford University, Stanford, Ca. 1979.

[VCWM77] vanCleemput, W.M., "An Hierarchical Language for the Structural Description of Digital Systems," Proc. 14th Design Automation Conference, New Orleans, 1977, pp. 377-385.

[WP76] Wilcox, P., and H. Rombeek, "F/LOGIC - An Interactive Fault and Logic Simulator for Digital Circuits," Proc. 13th Desian Automation Conference, San Francisco, 1976, pp. 68-73.

[WN75] Wirth, Niklaus, "An Assessment of the Programming Language PASCAL", IEEE Transactions of Software Enqineering, Vol SE 1, 2, June 1975.

APPENDIX C

INDEX