Stanford Artificial Intelligence Laboratory
Memo AIM-323

January 1979

Computer Science Department
Report No. STAN-CS-70-7 18

AL USERS' MANUAL

by

Shahid Mujtaba and Ron Goldman

Research sponsored by

National Science Foundation

COMPUTER SCIENCE DEPARTMENT
Stanford University

AL Users' Manual

by

Shahid Mujtaba & Ron Goldman

Stanford Artificial Intelligence Laboratory
Stanford University, California 94305.

January 19 7 9.

ABSTRACT

This document describes the current state of the AL system now in operation at the Stanford Artificial Intelligence Laboratory, and t eaches the reader how to use it. The system consists of AL, a high-level programming language for manipulator control useful in industrial assembly research; POINTY, an interactive system for specifying representation of parts; and ALAID, an interactive debugger for AL.

This work was supported by the National Science Foundation through grants NSF-APR-74-01390-A04 and NSF-DAR-78-15914.

TABLE OF CONTENTS

1. HOW TO USE THIS MANUAL

This document at tempts to gather in one place all the information that a user needs to program a manipulator in the AL programming language at the St anford Artificial Intelligence Laboratory (SAIL), In addition to meeting the requests made by other research organizations for detailed information on the current stat us and configuration of the system at SAIL, it updates the original AL document which was a design specification of the language. At SAIL, the most up to date version of the documentation can be obtained by the monitor command DO ALXGP.DOC[DOC,HE]. Specific chapters may be obtained by the command DO ALXGP[DOC,HE](n) where n is the chapter you are interested in. The AL system is growing and continuously evolving as new features are implemented and used. This edition of the manual includes features not present in the first edition of November 1977.

Chapter 2 describes the AL programming system and the related hardware and software configuration at SAIL, It is an overview and description for the general reader.

Chapter 3 and the succeeding chapters are for the AL user. Chapter 3 is an example based tutorial illustrating the use of simple AL instructions. This chapter assumes that the reader is familiar with interactive computer programming in high level languages such as FORTRAN. Previous knowledge of manipulator programming or programming in ALGOL is not essential. After completing this chapter, the user should have at his command a subset of AL instructions which will enable him to write simple programs.

Chapter 4 describes the AL language, and gives the complete set of currently implemented AL instructions in a concise manner. This chapter should be sufficient for an experienced programmer to use as a reference manual.

Chapter 5 describes how to execute AL programs.

Chapter 6 describes POINTY, a system which allows the user to generate interactively the frame tree data structure for AL programs,

Chapter 7 describes errors that might occur in the different stages of compiling and executing the AL program, and how to cope with them.

The Appendices include a list of AL reserved words and brief summaries of AL and POINTY commands and instructions, and extended AL and POINTY examples.

2. THE AL SYSTEM AT SAIL

2.1 Design philosophy of AL

2.1.1 Introduction and historv

The WAVE system for manipulator control was designed and implemented by Lou Paul in 1973 on the Scheinman Stanford model arm and was used extensively by him and Bob Bolles.

The experience with WAVE led to the initial specifications of AL in 1974 by both of them and Jerry Feldman, Ray Finkel, and Russ Taylor.

The initial implementation of the compiler and runtime system for Al was done by Finkel and Taylor, and subsequently taken over by Ron Goldman.

Vic Scheinman designed the arm currently in use at SAIL, while Tom Gafford and Ted Panofsky are responsible for the computer interface to the manipulator. Ken Salisbury and Gene Salamin are currently maintaining the hardware.

The work of Paul and Bruce Shimano was responsible for developing the kinematics of manipulation and the arm servo code. Shimano subsequently implemented force compliance, while Tatsuzo Ishida has done a theoretical analysis of two arm cooperative manipulation. Salisbury is currently maintaining the arm servo code.

ALAID, for debugging AL programs, was initially implemented by Finkel. A newer version is being implemented by Hamid Nabavi.

The first AL parser was written by Bill Laaser and Pitts Jarvis, and subsequently taken over by Shahid Mujtaba.

POINTY, a related system, was conceived in 1975 by Dave Grossman and Taylor, and init ialiy implemented by Taylor. Maria Gini, Pina Gini and Mujtaba have subsequently implemented a newer version. Enrico Pagello has also contributed to it. POINTY is currently maintained by Mujtaba.

The design of AL has continually been modified and updated on the basis of new experience and information by Grossman, Shimano, Goldman, and Mujtaba under the overall guidance of Tom Binford.

The AL system is geared towards batch manufacturing where setup time is a key fact or. To minimize programming time we rely on a symbolic data base and previously defined assembly primitives, and a quick and simple means of putting into a program the things we want to tell the manipulator to do. By testing out the system on undergraduate industrial engineering students with minimal experience in manipulators and robotics, we have found that learning to use the AL system is relatively simple, and that it is unnecessary to learn the complete system before putting it to use. Team programming sessions by researchers in manipulation at the Workshop on Software for Assembly held in November 1977 at SAIL showed the possibility of learning to program AL in a short time. The AL system has also been used for term projects in a Robotics course given in the Fall quarter of 1978.

We assume that the batch manufacturing environment is fairly structured and known - the positions of fixtures, parts, tools, etc. are known and not expected to vary from one assembly to another by any appreciable amount. By simulation it is possible to predict where each object is at any instant of time, and whether it is held in a manipulator, if it is to be moved, and whether sufficient information has been given, so that communication with the user is minimized during execution, Because of the high degree of structure in the manufacturing environment, we try to do as much computation as possible before an assembly begins.

2.1.2 Plantime and runtime systems

Experience with WAVE (the predecessor to AL) had shown that calculating trajectories for manipulators was a desirable feature. It was thus decided that trajectory calculations, together with ail other calculations which need only be performed once, should be done at compile time on the assumption that this allocation of effort would reduce the computing load at execution time and eliminate recomputation every time a sequence of actions is executed.

This sequence of planning and execution led to the existence of two systems - the plantime system and the runtime system.

The plantime system consists of the AL compiler whose function is to take the user written AL program, simulate it, point out errors to the user, and output instructions to the runtime system. The compiler performs a simulation of the program (called world modeiling) to verify that it is indeed possible to do what the user asks within the limits of what AL is capable of doing, and to warn the user about unexpected consequences (e.g. if the user accidentally asks that the arm be moved through the table). The runtime system takes the output of the plantime system, and proceeds to perform the motions.

This approach is changing because of subsequent developments. Computation costs have dropped dramatically and this makes possible the future use of multiple processors in distributed computation. Better arm servo software, faster arrn solution and more sophisticated path calculation algorithms tend to reduce the computation load, thereby permitting more decisions to be made at runtime. It was also realized that certain trajectories are best computed during runtime (e.g. force compliant motion, moving belt, when the workplace is highly unstructured). (See "Discussion of Trajectory Calculation Methods" by Mujtaba in Progress Report 4.).

2.1.3 Data and control structure

The principal mode of input to AL is textual. The use of symbolic programming means that for parts in a pallet, for instance, there is no need to define the position of all the parts, if the distance between parts (which is usually constant) is known. "Once the corner of a pallet is taught and the part separation is known, laborious record-playback programming is no longer necessary given proper software in an associated minicomputer," [Engelberger, J.F. in "A Robotics Prognostication", Joint Automatic Control Conference Proceedings 1977, p 198.1 Symbolic programming simplifies the interfacing of AL with other means of generating world models, like interactive graphics and computer aided design. It permits the setting up of library programs which may be called by supplying the relevant parameters. The use of symbolic programming eases the job of specifying complex motions if such motions can be parametrized or described algebraically - for example, it is easier to tell the hand to move a certain distance along an arbitrary direction than it is to move it manually when multiple joints have to be adjusted simultaneously. Teaching by doing, on the other hand, requires the recording of a very large number of points (tape recorder mode) unless only the end points of motions are of interest and the nature of the paths between these end points are unimportant.

There are levels of complexity which are much more readily transmitted from man to machine through an interface of symbolic text. Simultaneous motions of two arms, specifications for termination, and error conditions are more likely to be unambiguously stated through the medium of text, since these may require multiple logical relationships to be satisfied. Non-textual forms of input can be a very useful rneans for defining target locations, suggesting arm trajectories designed to avoid collisions, initial setup of a workstation, and other purposes of this nature.

AL makes use of more data types than other conventional high level languages do. In addition to *SCALAR* numbers, it allows the specification of *VECTOR*, *ROTATION*, *FRAME*, *TRANS*, and *EVENT* data types. A *VECTOR* consists of a triple of three real numbers. A *ROT* consists of a direction vector and an angle to indicate the amount of rotation. A *FRAME* describes the position and orientation of an object, while a *TRANS* describes the relationships between FRAMES. In addition, arrays of all these data types may be defined. Arithmetic operators are available not only for the standard scalar operations but also for such operations as rotation and translation.

We want to write programs in a natural manner. The machine-language like aspect of current manipulation languages makes it cumbersome to write long programs in any structured way. We want a language which lends itself to a more syst ematic and easily understood programming style. To this end, the use of ALGOL-li ke cont roi structures are an improvement over linear machine-language code with jumps. The block structure of ALGOL is also present in AL.

Experience with languages like SAIL and WAVE has shown that text macros are a useful feature; they reduce the amount of repetitive typing, and allow symbolic definition of constants and variables in a way which would be otherwise impossible. AL has a general-purpose text macro system.

Procedures are provided, as in other languages, to reduce the amount of code when similar computations or operations need to be done at numerous places in a program.

AL also permits the control of parallel processes by allowing the flow of control of the program to be divided up, which allows certain operations to be performed simultaneously (e.g. simultaneous movement of different manipulators), after which the various processes merge back together. Synchronization primitives are also provided.

2.1.4 Motion of objects

AL has a mechanism to keep track of the location of a component piece of an assembly automatically as the assembly is moved; this mechanism is called affixment and used extensively with the concept of *FRAME* to describe objects. Frames may be affixed to each other, so that after affixing an object to the manipulator, the user can forget about the manipulator completely, and think in terms only of where objects have to interface with other objects. instead of having to worry about how to move the arm, the user can specify the final orientation and position of the object, and AL will take care of working out what the arm has to do in order to accomplish the stated objective. The user can think of movement of the objects rather than the movement of the manipulator. This is significantly different from other programming schemes where the program consists of a series of arm motions whose relationship to objects in the real world is known only to the user, and where the user effectively has to provide explicitly the distance and angular relationship of the object to the manipulator for each motion statement,

2.1.5 Sensory informat ion

AL allows alternative actions on the basis of sensory input during runtime by checking whether certain conditions have exceeded a specified threshold, and if so to execute a predetermined action. This is called condition monitoring. When error conditions are encountered, it is possible to set a sequence of actions into motion that will try to allow recovery. This is not possible in the case of linear control where program execution has to be aborted.

2.1.6 Programming aids

AL has several features that help the user during different phases of compilation and execution of his program to ensure that errors are caught as early as possible, and to simplify programming.

2.1.6.1 AL parser

The AL parser takes the user-writ ten AL program and checks that it is syntactically correct, generating error messages where necessary. It also makes use of the AL declarations generated by POINTY if told to do so. It enables programs to be input through disk files written by means of text editors, or through the teletype.

The AL parser tries to catch and correct errors early, so that less time is wasted on a compilation if it needs to be aborted. Also, by catching errors early, it is possible to generate error messages in the context of the user source program. Two main checks are used to eiirninate an important class of errors. Dimension checking across assignments and expressions is done by the parser to ensure that units have been correctly specified by the user and are compatible with what is expected. Type checking across assignment statements and across the terms and factors of an expression ensure that operations are performed on arguments of the right data type, and that assignment of an expression is done to a variable of the same data type.

AL allows interactive error correction by permitting the user to ask for a standard fixup, or to change (patch) the offending source code for minor errors and continue from there without having to resort to the system text editor and a recornpilat ion. Error recovery is local, and permits backing up to the beginning of the innermost current statement. To back up any further, it is necessary to resort to the text editor. At the user's option, a corrected copy of the source file is made,

2. 1.6.2 AL compiler

The AI compiler provides a number of semantic checks on the user's program. Warnings will be issued if an attempt is made to move the arm to an inaccessible location (e.g. through the table top), if not enough time was allocated for a mot ion, if incompatible force requests are made, if the user attempts to move something not connected to an arm, or if the arms are not parked upon program completion.

To help track down errors the user can request that the planning value AL maintains for each variable during the world modelling phase of compilation be printed at some desired point in the program.

2.1.6.3 Interactive model building

POINTY, to be described in detail in Chapter 6, allows the user to create interactively the frame tree for AL programs with the aid of the manipulator as well as to try out simple motion statements, The interactive nature of POINTY is also helpful in testing out small segments of programs before incorporating them in a larger AL program.

2.1.6.4 Debuggers

Several debuggers are available during execution of the program to enable the user to correct his mistakes by allowing him to patch his programs, and to let him examine and change the values of variables.

Debugging an AL program during execution involves examining and modifying variables, altering the flow of control, triggering condition monitors, and patching code. ALAID has been designed to permit these actions to be performed and to assist the programmer in preparing correct manipulator code. ALAID sets up a communication link between the PDP-10 and PDP-l 1 (cf. section 2.2) and allows debugging to proceed from either machine, It sets up a clean interface between an AL program running on the POP-1 1 and a higher level strategy program on the PDP- 10. ALAID enables the two processes to signal each other using the synchronization primitives in AL and it also allows the program running on the PDP-10 to examine and set variables in the memory space of the AL manipulator program on the POP-l 1.

In its current state, ALAID connects the two machines, can examine and set variables, signal and wait for events, cause the runtime system to enter 1 1 DDT, allow the user to stop and resume execution of an AL program on the PDP-11, and examine and modify the pcode.

1 1 DDT is the PDP-11 machine language symbolic debugger used by AL

wizards to debug the runtime system, and by the user to continue or restart execution of his program.

2.2 AL system hardware

The hardware for the AL system consists basically of a PDP KL10 computer (hereafter referred to as PDP-10) for compiling and loading the AL program, a PDP-11/45 computer (hereafter referred to as PDP-11) for executing the AL program, and two STANFORD model Scheinman arms in addition to various peripherals such as terminals and disks.

The relationships between the various components are shown below. The PDP-1 1 system is interfaced to both the manipulators and the PDP-10 system. Any communication between the PDP-10 and the manipulators must go through the PDP-11 runtime system, since there is no direct interface between the PDP-10 and the arms.

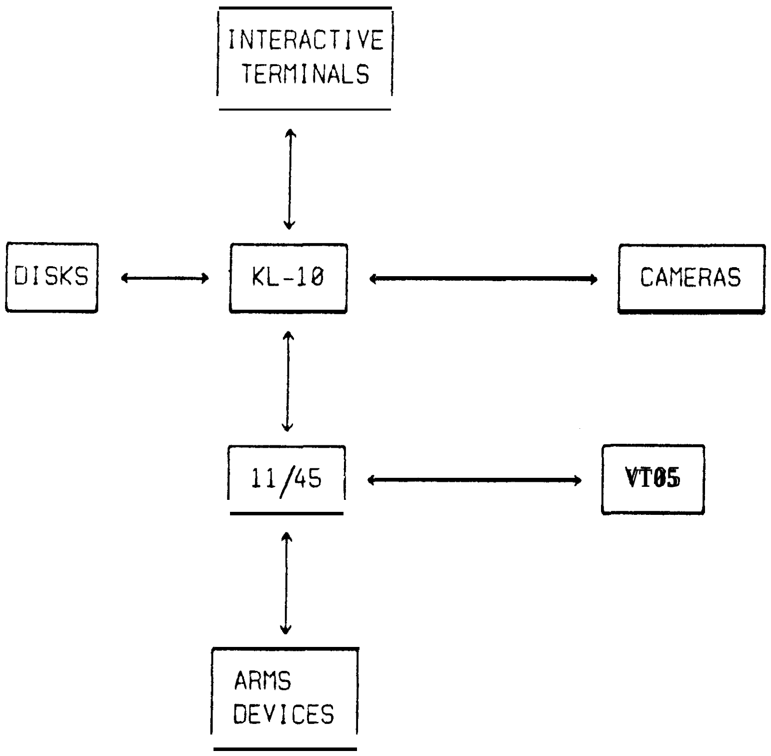


Fig. 2.2 Hardware setup for AL at SAIL

2.3 Soft ware

The software organization of the current AL system at SAIL is shown in Fig. 2.3. Each of the blocks indicates a module of programs that can be in core at one time, and the files that each module needs and generates are written alongside the lines connecting the modules.

Data and programs are stored in files which have names of the form ABCDEF.XYZ where ABCDEF is a combination of one to six alphanumeric characters making up the name, and XYZ is any combination of zero (in which case the dot is omitted) to three characters, making up the extension. The extension serves to distinguish different files in a family of files of the same name.

Affixment information can be generated in AL statement form using POINTY and saved in a declaration file. The motion program and data can be prepared and saved on a disk file FOO.AL (where FOO is the name, and .AL is the extension which serves to identify an AL source program) using the text editor. It can also be input directly to the AL compiler through the teletype.

The AL parser takes the AL program written by the user, and checks that it is syntactically correct, generating error messages where necessary. It generates an intermediate file (using S-expressions), with extension SEX, that is passed to the AL compiler. At the option of the user, the AL parser will generate a logging file with extension .LOG with all the error messages, and a corrected copy of the source file with extension .NEW. For programs input directly through the teletype, a disk file copy of the program will be generated with extension .TTY.

The AL compiler reads in the S-expression file generated by the AL parser, and changes it into an internal form. It then performs a simulation of the program called world modelling, followed by trajectory calculation and code generation. At the end, four files with extensions .ALP,.ALV,.ALT,.ALS, are emit ted, which contain information on the pcode, numerical constants, motion trajectories, and symbols, respectively.

The first three files are used by PALX, the PDP-11 cross assembler to assemble a binary load module having extension .BIN for the runtime system.

The binary file with extension .BIN is loaded together with the AL pcode interpreter and the run time system by a program called 1 1TTY.

The intermediate files with extensions .ALP,.ALV, and .ALT are typically deleted by the ALSOAP program, which is run automatically after the AL compilation.

If so desired, ALAID can be run simultaneously on the PDP-10 to provide a communication link between the PDP-10 and PDP-11. ALAID makes use of the .ALS file.

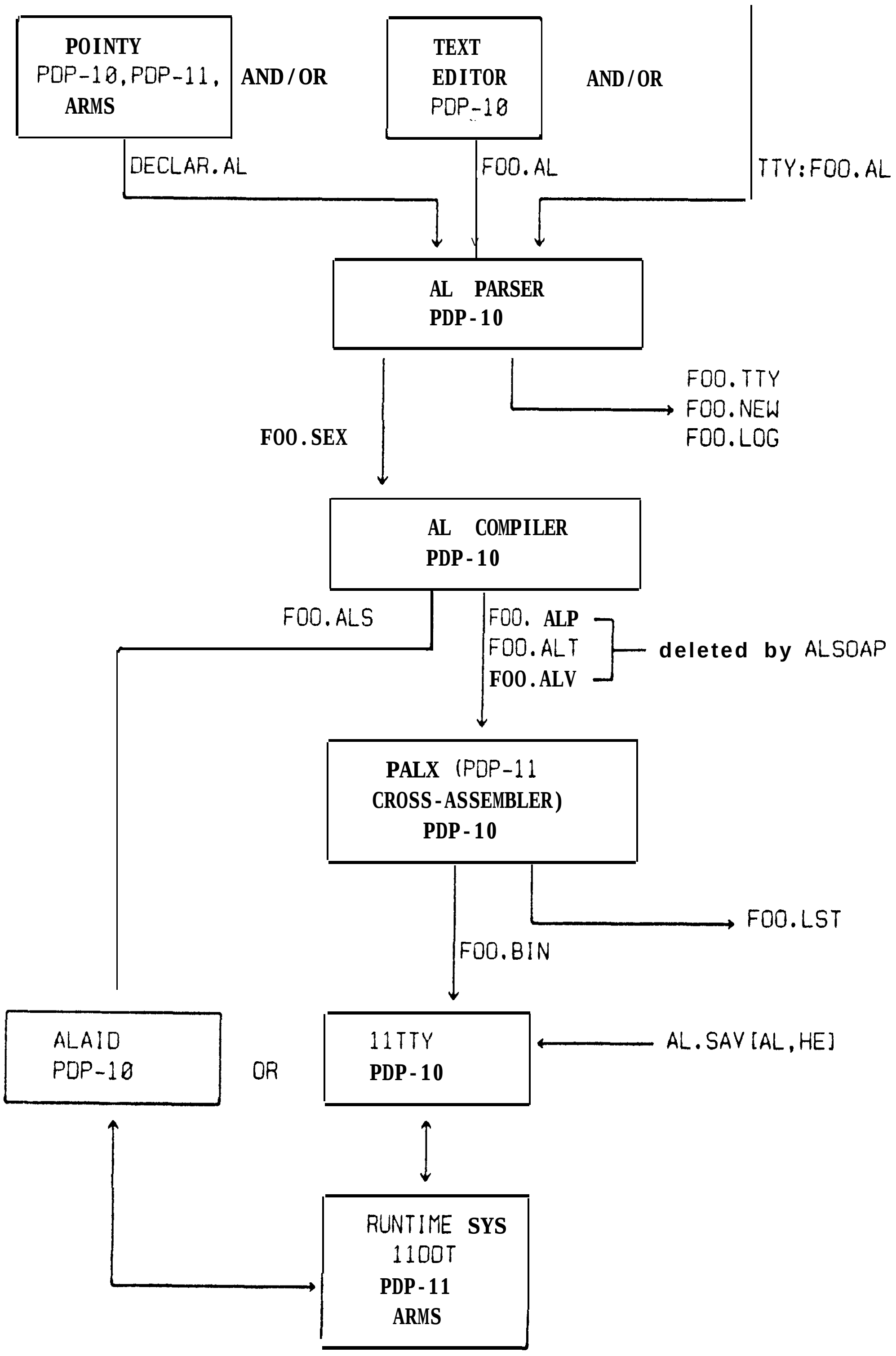


Fig. 2.3 Software organization

2.4 Programming in AL

In order to program an assembly in AL, an assembly plan should first be worked out which includes a rough layout of the parts, and the sequence of motions to accomplish the assembly.

The parts and fixtures should be laid out in the work place in the desired physical locations. AL has to be given the information of the object layout, and this can be done either by direct measurement using a ruler and other measuring equipment, or with the aid of manipulators and POINTY, an interactive data gathering aid using the manipulator (c.f. Chap 6). The data must be incorporated into a file which has AL statements which specify how to move the parts to accomplish the desired assembly.

Having obtained the program, the user gets it into the computer system by some means (at SAIL this means through one of the interactive terminals).

The program is compiled, loaded, and executed and debugged much like any other program.

3. USING THE AL SYSTEM

3.1 Basic constructs

The purpose of this chapter is to introduce the reader to the AL language, and through a series of examples, show its use in the programming of manipulator motions. The basic constructs of the AL language are described in this section. Other instructions will be described in the the following sections of this chapter.

The notation will be as follows; Within the programs and examples reserved words will be shown in upper case, while variables and predefined constants will be shown in lower case. In all other places, they will be represented in upper and lower case italics respectively.

3.1. 1 Data types

At the heart of each computer language are the types of data that can be handled. For example, FORTRAN has INTEGER and REAL numbers; other languages can handle strings of alphabetic characters. The data types in AL were chosen to handle the special problems that arise in controlling manipulators, and in working with three-dimensional objects in the real world which have directed distances, locations and orientations.

A variable is an identifier that can take on various values. Identifiers consist of a string of alphanumeric characters (letters and numbers) and underscore "_". Some examples: *pump_base, handle, screw_hole_2,* and *P132.* Note that all identifiers must start with a letter (*3inch_screw* is no good). Upper and lower case are equivalent, i.e. *ABC, abc,* and *aBc* all refer to the same variable.

Variables can be given a value by means of an assignment statement, which consists of the variable name, a left arrow ("←"), and an expression of the correct type. When an assignment statement is executed, the expression on the right hand side is evaluated, and the result becomes the new value of the variable on the -left hand side.

AL, like ALGOL, requires each variable to be declared, that is, one must state what data type a variable is before it is used. AL also uses ALGOL type block structure which means that all variables declared between a particular *BEGIN* and *END* are accessible only to code which appears between the same *BEGIN-END* pair. It is also possible for the same variable name to be used in different blocks without conflict. Block structure will be explained more fully later (3.1.2). We shall now look at the data types available in the AL language.

3.1.1.1 *SCALARS*

The most elementary data type in AL is the SCALAR, which is internally represented as a floating-point number. Scalars can be used for dimensionless quantities, such as the number of times some operation is to be repeated, or for dimensioned ones like the length of an object or the angle between two parts. The arithmetic operations available on scalars are addition, subtract ion, multiplication, division and exponentiation, represented by the normal arithmetic operators: "+", "-", "*", "/", and "↑". Exponentiation has precedence over multiplication and division which in turn have precedence over addition and subtraction, as in other algebraic languages. Several commonly used functions are also available: the square root function, *SQRT*; the trigonometric functions SIN, *COS*, *TAN*, *ASIN*, *ACOS* taking one argument, and *ATAN2* taking two arguments; the natural logarithm *LOG*; and the exponential function *EXP*.

Scalar constants are written as (base ten) numbers, possibly with a decimal point or fractional part; for example 2, 1., 3.14159, -123.45 are all scalar constants.

Below is an example showing the declaration and use of scalar variables. In the examples in this section we will use a mnemonic scheme for naming variables to clarify the type of each entity. Note that AL statements are separated by semicolons. Also curly brackets "{}" are used to enclose comments.

```
SCALAR sl, s2;       (A declaration consists of a data type followed by
                     a list of variable names separated by commas, and
                     ending with a semicolon.}

sl ← 2;
s2 ← 3.50;                {s1 has the value 2.0, and s2 is 3.50)
sl ← s2 * (s1 - 3.2);     {Now sl = -4.20}
```

It is often desirable to associate a physical dimension with a variable. AL provides for scalars with the dimensions of *TIME*, *DISTANCE*, *ANGLE*, and FORCE. It should be noted that *ANGLE* is generally considered dimensionless, but that for our purposes, the definition has been made a little flexible to allow for an entity which is useful for defining rotations. Dimensioned variables are just like regular scalar variables, except that they are associated with an appropriate dimensional unit: *sec*, inches, *deg* or *ounces*, which have the obvious meanings. AL can also handle CM, *oz*, *lbs*, *gm* and *radians*.

Dimensioned variables are used exactly in the same way as simple variables, except that AL checks for consistent usage. Dimension checking is done for each arithmetic operation and each assignment. Addit ion, subtract ion and assignment require exact dimensional match, though if the match fails and one of the two arguments is simple (dimensionless), it will be coerced to the right type, after an appropriate message to the user. Multiplication and division do not require dimensional match; they produce a result of a dimension different from that of the arguments which is then propagated through the expression. In this way intermediate results can be of dimensions not declared. This causes no problems unless such results are used in an assignment; The square root function may be used on scalars of arbitrary physical dimensions, and the dimensions of the result will be the square root of that of the argument. The *SIN, COS* and *TAN* functions are applied to scalars having dimensions of *ANGLE* and assumed to have units of degrees. The result is dimensionless. The inverse functions *ASIN*, *ACOS*, and *ATAN2* take dimensionless arguments; the resulting value has dimensions of *ANGLE* and units of *DEGREES*. The exponential and logarithmic functions take dimensionless arguments and return dimensionless values. The exponentiation operator presents a problem for the parser, since during parsing, the value of the power to which the base is raised is unknown. The problem is recognized by giving an error message if either the base or index is not dimensionless.

Here is a short-example using dimensioned scalars and functions.

```
SCALAR sl, s2;
DISTANCE SCALAR dsl;
TIME SCALAR tml, tm2;
FORCE SCALAR fsl;
ANGLE SCALAR theta, phi;

dsl ← 1.0 * inch;
trnl ← 3 * sec;
fsl ← 2.2 * ounces;
tm2 ← tml + 4.5;          {The constant 4.5 will be converted to
                           seconds after the relevant error message.}
theta ← 90 * deg;
phi ← theta * 4 * deg;    {This is a mistake: the right hand side has
                           dimension ANGLE * ANGLE.}
s1 ← SIN( 30 * deg);
theta ← ACOS(.7);
ds1 ← SQRT(dsl * 3 * inches );
phi ← ATAN2( sl, s2);     { same as arctangent(s1/s2) }
sl ← LOG(33.0);
s2 ← s1 ↑ 3;
```

There are several predeclared scalars in AL:

```
SCALAR PI;                {3.14159...}
```

π is also recognized as the constant 3.14159....

```
DISTANCE SCALAR bhand, yhand;
                       {These variables refer to the blue hand and
                       yellow hand openings)
```

VELOCITY, *ANGULAR_VELOCITY*, and *TORQUE* are defined in terms of the primary dimensions in the generally accepted way,

It is also possible to define new dimensions, such as *acceleration*, by means of the dimension statement. New dirnensional units, such as *feet*, can be defined with macros (4.5.8). For inst ance:

```
DEFINE feet = ⊂( 12 * inches)⊃;
DIMENSION acceleration = VELOCITY / TIME;

acceleration SCALAR as1;
as1 ← 6.7 * feet /(sec * sec); {= 6.7 * 12 * inches/sec/sec}
```

3.1.1.2 *VECTORS*

The world in which AL programs operate has three dimensions, and so we need rnore than just scalars. We will now introduce another data type: the *VECTOR*. it and the other algebraic data types which follow are similar to scalars in how they comprise arithmetic expressions and assignments.

We describe the world as a Euclidean space with three cardinal orthogonal axes, which meet at an origin. The actual alignrnent of these station axes is implementation dependent, though at SAIL and for the rest of this manual it will be assumed that the positive Z axis points upwards.

Vectors may represent entities having both direction and magnitude, e.g. displacement, velocity, acceleration, Like scalars, they may be dimensioned. Vectors can be constructed fron three scalar expressions by means of the function *VECTOR*. The scalar expressions must all be of the same dimension, which the resulting vector will also have.

The available operations between vectors include addition, subtraction, dot product, and cross product. A vector may be multiplied or divided by a scalar. The direction unit vector (dimensionless) may be extracted by the function *UNIT*. Addition and subtraction are defined only on vectors of the same dimension. The dot product, cross product and multiplication by a scalar give results having the dimensions which are the product of the dimensions of the two arguments. The scalar magnitude of a vector is obtained by enclosing it within vertical bars. The operators are defined in the normal manner; for example, if we have a scalar *s* and two vectors:

```
v1 = VECTOR(x1, y1, z1) and v2 = VECTOR(x2, y2, z2)
```

then we have:

```
s * v1 = v1 * s = VECTOR(s * x1, s * y1, s * z1)
v1 + v2 = VECTOR(x1+x2, y1+y2, z1+z2)
v1 - v2 = VECTOR(x1-x2, y1-y2, z1-z2)
v1 . v2 = x1*x2 + y1*y2 + z1*z2
```

There are several predeclared vectors in AL:

```
VECTOR xhat, yhat, zhat, nilvect;        {These have values as follows}
xhat ← VECTOR(1,0,0);
yhat ← VECTOR(0,1,0);
zhat ← VECTOR(0,0,1);
nilvect ← VECTOR(0,0,0);
```

Here is one more example of the use of vectors:

```
VECTOR v;
DISTANCE VECTOR dv1, dv2, dv3;
SCALAR s;
DISTANCE SCALAR ds1, ds2;

ds1 ← 2 * inches;
dv1 ← VECTOR(4, 2, 6) * inches;
ds2 ← dv1 . yhat;                   {So ds2 = 2 * inches}
v ← VECTOR(2, 1, 3);
v ← v - zhat;                       {So v = VECTOR(2, 1, 2) }
dv2 ← VECTOR(3, 0, 4) * inches;
ds1 ← |dv2|;                        {This assigns ds1 the magnitude of
                                    the vector dv1, which is a scalar of
                                    the appropriate dimension. So ds1 = 5
                                    * inches.}
dv3 ← VECTOR(4*inches, 2*inches, 6*inches); {dv3 is the same as dv1}
v ← UNIT(v);                        {So v = VECTOR(2/3, 1/3, 2/3) }
```

3.1.1.3 *ROTATIONS*

The next data type we will discuss is the rotation, or *ROT*, which represents either an orientation or a rotation about an axis. Rotations can operate on vectors and rotate them around the origin (without changing their length). They can also operate on other rotations (by matrix multiplication). To rotate a vector (about the station origin), multiply the vector (on the right) by the rot (on the left). To compose rots, multiply them together; the one on the right will be applied first.

The axis of rotation can be extracted by the function *AXIS* and the angle of rotation by enclosing the rotation expression within vertical bars. Rotations are dimensionless, and the user may not specify dimensions for this data type; however the arnount of rotation about the axis has units of *ANGLE*.

A rotation can be constructed with the function *ROT*, which takes two arguments: a simple vector, which is the axis of rotation, and an angle, which is the amount to rotate. The direction of rotation follows the right hand rule, so a rotation of 90 degrees about the X axis moves the Y axis into the Z axis. This representation is far easier to write and understand than raw matrices. Here is an example showing the use of rotations:

```
ROT r 1, r2, r3, r4;
ANGLE SCALAR alpha, beta, gamma;
VECTOR v;
rl ← ROT(xhat, 90 * deg);
v ← r 1 * zhat;        {v gets Z rotated 90 degrees about X, so v =
                       VECTOR(0,-1,0)}
r 2 ← ROT(yhat, 4 5 * deg);
r 3 ← r2 * rl;
     {Thus, r3 means first rotate 90 degrees about the X axis, then
     45 degrees about the original Y axis.   An alternative
     interpretation is to first rotate by 45 degrees about Y, and then
     to rotate by 90 degrees about the new X axis.}
v ← AXIS(r2);          {This assigns v the axis of rotation of r2 = yhat.}
alpha ←|r2|;           (This assigns alpha the angle of rotation of r2 = 45
                       degrees.}
rl ← ROT(xhat, alpha);
r2 ← ROT(yhat, beta);
r3 ← ROT(zhat, gamma);
r4 ← r3 * r2 * r1;
     {r4 is then a rotation with the following two meanings: Rotate
     by alpha degrees about the X axis of the station, then by beta
     degrees about the station's Y axis, and finally by gamma
     degrees about the station's Z axis. Or alternatively, rotate by
     gamma about the station's Z axis, then by beta about the new Y
     axis, and finally by alpha about the doubly new X axis. Both of
     these interpretations yield the same result; use whichever one
     you find most comfortable.}
```

There is one predeclared rot, called *nilrot*, defined as *ROT(zhat, 0 * deg)*.

3.1.1.4 *FRAMES*

In working with objects in the real world we need to specify both their position and orientation. To do this we introduce a new data type, the *FRAME*, which represents a coordinate system. It has two components: the location of the origin (a dist ance vector) and the orientation of the axes (a rot). Features on an object can be specified with respect to the object's coordinate system.

There are several predeclared frames in AL. *Station* represents the reference frame of the work station. Associated with each manipulator is a frarne whose value (updated at the end of each motion} is the position of that rnanipulat or. Currently, there are two such frames: *barm* and *yam,* associated with the blue and yellow arms respectively. Also associated with each arm is a rest, or park position; these are *bpark* and *ypark.*

A frame may be constructed by calling the function *FRAME,* which has two arguments: a rot (for the orientation) and a distance vector (for the position). The orientation or position of a frame can be extracted by the functions *ORIENT* and *POS.* To transform a point specified by a distance vector in the coordinate system of sorne frame into station coordinates, multiply the frame (on the left) by the vector (on the right). To translate a frame by some amount, simply add/subtract a distance vector to/from it. Finally, to construct a vector in station coordinates which has the same orientation as a vector in some frame, such as *xhat* in say *fl*, the "with respect to" operator *WRT* is used and one writes *xhat WRT fl.* For any vector v and frame *f* the following are equivalent (the dimensions of the result are the sarne as those of *v*):

$$v \text{ WRT } f \equiv (f * v) - \text{POS}(f) \equiv \text{ORIENT}(f) * v$$

Here are a few examples using frames.

```
FRAME f1, f2;
f1←FRAME(ROT(zhat, 90 * deg), 2 * xhat * inches);
        (The frame f 1 sits 2 inches from the station in the X direction,
        Its coordinate system has X where the station's Y axis points,)
v1← xhat WRT fl;          (This evaluates to VECTOR(0,1,0).}
f2 ←f1+v1* inches;        (Just like fl, but with origin at (2,1,0).}
v2←f1*(zhat* inch);       {This evaluates to VECTOR(2,0,1).}
```

3.1.1.5 *TRANSFORMS*

The last of the algebraic data types is the transformation or *TRANS*. Transes are used to transform frames and vectors from one coordinate system to anot her, Like frames, they consist of two components: a rotation and a vector, The application of a trans first rotates its operand about the station origin, and then translates the result. Transes can be composed in the same manner as rotations, the one on the right being applied first.

A t rans consists of a rotation part having units of angle and a translational (vector) part having some other physical unit - usually distance. When "multiplying" by a trans, one is really multiplying by the rotational part and then adding the vector component. The matrix operation of multiplying transes together produces a trans. The vector parts of two transes multiplied toget her must have the same dirnensions, and the vector part of the product will have the same result. For convenience, we will refer to the dimension of a trans as being that of the vector part. When a trans is applied to a vector, both must have the same dimension, the one for the trans being defined above. The resulting vector is of the same dimension, When a t rans operates on a frame, it must be a dist ance trans. When transes are composed, they must agree in dimension, and the result will have the same dimension, Unless declared otherwise, transes will be assumed to have dimensions of distance.

One can construct a transform by use of the function *TRANS*, which takes two arguments: a rot (the rotational part) and a vector (the translational part). Another convenient way to specify a trans is by forming it from two frames. The arithmetic operator "→" applied to two frames produces a trans which takes the origin of the first frarne across to the origin of the second, performing a rotation first to get the axis aligned. When a frame is used in a context dernanding a transformation, it will be understood as a shorthand for the distance trans leading to it from the station.

Here are a few examples using transes.

```
TRANS tl, t2, t3, t4;
t 1 ←TRANS(ROT(xhat, 60 *deg), 2 * zhat * inches );
v1←t 1 * yhat * inches;
                        {t 1 rotates yhat 30 degrees about the X-axis, and then
                         translates it by 2 inches along Z = (0,.866,2.5).}
t2 ← fl → f2;           (Thus f1* t2 = f2.}        .
v2← t 2 * (xhat * inches);
                        {v2 is f2's x-axis as seen from fl}
t 3 ← t2 * t1;          {t3 means to first perform the transformation given by t 1,
                         and then that specified by t2.}
f3 ← fl * f2;           (This expresses the position of f2 in f 1's coordinate
                         system. Equivalent to (station →f1)*f2.}
t5 ←INV(t 1);           (This expresses the inverse transformation of t 1.)
```

The null transformation, equivalent to *TRANS(nilrot,nilvect)*, is called *niltrans.*

The initial distinction between frames and transes has lessened as work with AL has progressed. The current distinction is that frames may be affixed to each other, and have deproach points (c.f. section 3.4) associated with them. In general a trans can appear anywhere a frame can, and vice versa. For example to get at either of a trans's two components the extraction operators, *ORIENT* and *POS*, would be used. Whether or not the two data types will be merged rernains to be seen. An evolving view considers frames to be labels associated with physical objects or locations in space and transes the relationship between these physical objects, In such a case, frames would not have dimensions associated with them, but there will be some relationship between them and other frames.

3.1.2 Block structure - i.e. "what's a program"

An AL program consists of a sequence of statements which will result in the manipulator successfully performing a desired task. While the simplest AL program consists of a single simple statement, any reasonable program will be made of many statements *S1, S2, S3, . . .* separated by semicolons, and surrounded by the reserved words *BEGIN* and *END*. This composite arrangement of

```
BEGIN
   S1;
   s2;


   Sn
END
```

is known as a block statement. The statements *(S1,S2,...)* within the block may themselves be other block statements. Indentation has no effect on the program and serves only to make the program more readable.

In order to keep track of blocks within other blocks, they may be named with strings within double quotes immediately following the *BEGIN* and the corresponding *END*, The strings after a corresponding *BEGIN* and *END* pair should be the same, or there should be no string after the *END;* otherwise there will-be an error message. The following is an example of block naming:

```
BEGIN "MAIN"
    S1;
    s2;
    BEGIN "INNER"
      S3a;
      S3b;
    END "INNER";
    s4
END
```

Like SAIL or ALGOL, AL requires that an identifier be declared before it is used. The effect of an identifier is only within the block it is declared. Outside the block, any reference to those identifiers will give an error message, An error message will result if the same identifier is declared more than once in a given block, unless subsequent declarations are within blocks internal to the given block. Consider the following example:

```
BEGIN "BLK_1"
   SCALAR i,k,m;
   i←1;
   BEGIN "BLK_2"
      SCALAR i;   {denotes a new variable "i" distinct from
                        the "i" declared in BLK_1 above}
      i←2;
      m←i;        {So m=2; i refers to the second declaration of i}
   END "BLK_2";
   k←i;           (So k=l since after exiting "BLK_1" i=1 again}
END "BLK_1";
```

In the inner block "*BLK_2*" the variable *i* is a new variable distinct from the *i* defined in "*BLK_1*". Had the *SCALAR i* statement been absent in block "*BLK_2*", the value of *k* and *i* at the end of execution would have been 2.

3.1.3 A simple program

As mentioned before, an assignment statement consists of a variable name, a left arrow ("←"), and an expression of the correct type. When an assignment stat ement is executed, the value of the expression on the right hand side is computed, and the result becomes the new value of the variable on the left hand side. Care must be taken to ensure that the data type of the expression is the same as that of the variable. During compilation, AL will check for type and dimensional consistency across opposite sides of the left arrow, and complain if it finds any incompatibility.

The print statement prints out the values of the variables and the strings during execution time. It consists of the reserved word *PRINT* followed by an open parenthesis, a list of arguments separated by commas and a close parenthesis, The arguments may be variable names or the names of predefined constants, or they may be string constants which consist of characters enclosed by double quotes.

Here is a simple AL program that will compute and print out the current arm positions and the distance between them;

```
BEGIN
   DISTANCE SCALAR s1;
   DISTANCE VECTOR vl ;
   PRINT ("THE BLUE ARM IS AT ", barm);
   PRINT ("THE YELLOW ARM IS AT ", yarm);
```

```
        v1 ← POS(barm) - POS(yarm);
                              {v1 is tho vector distance between tho centers of
                              t ho hands}
        s1 ← | v1 |;          {s1 is the absolute distance between the hands}
        PRINT ("THE DISTANCE BETWEEN THE BLUE AND YELLOW FINGERS IS",
           s1 , " INCHES");
     END
```

Other statements possible within a block will be discussed in the following sect ions.

3.2 Simple *MOVE* statement

The simplest motion program is one which will move an arm to a known position. When the two arms *barm* or *yarm* are not being used they are placed in statically balanced positions with the fingers pointing downwards so that a power failure does not result in the arms collapsing. The resting positions of the arms with the described pointing direction (orientation) of the fingers are known as *bpark* and *ypark*.

For purposes of this document when we refer to an arm we shall mean the blue arm unless otherwise obvious from the context.

Let us assume that the arm is in any arbitrary posit ion, and we want to move it to the park position under computer control, The statement to do this would be

```
MOVE barm TO bpark;
```

During compilation, AL will try to work out a trajectory (the position of each of the joints from the initial position to the final position as a function of time) from the current position to the park position so that the motion is accomplished as fast as possible subject to the constraints of maximum acceleration and torque imposed by the motors. However, during compilation, AL cannot read the arm position, so it has to be provided with a planned position for the arm which the user may specify. Unless told otherwise, AL will assume at the beginning of a program that the arm is at t-he park position. During execution, if the actual position is different from the assumed starting position, the runtime system will try to modify the trajectory to accomplish the motion within the length of time originally planned. Thus if a joint has to go a distance further than originally planned, the motion would have to be faster than planned in order to be accomplished in the same time.

In the above statement, if AL assumes that *barm* is already at the park position, it will very wisely decide that no motion is required, and will thus compute a trajectory which requires *zero* time to traverse, Should it happen at execution that the arm is not initially at the park position, the modified trajectory will try to bring the arm to the park position in zero time, which will result in large accelerations and excessive motor torques being required. In order to slow down the motion to ensure its success, we should use a *DURATION* clause to modify the planned zero time trajectory as follows:

```
MOVE barm TO bpark WITH DURATION = 4*seconds;
```

This statement tells the computer that we want to move to the park position over a time interval of four seconds. Note that there is now no semicolon after the *bpark* but rather that it is at the end of the entire move statement after the *DURATION* clause.

It is also possible to specify differential motions. The grinch sign, "⊗", is used to represent the current position of the arm. The following statement would cause the arm to move down 2 inches.

```
MOVE barm TO ⊗ - 2 * zhat * inches;
```

3.2.1 More about *barm* and *bpark*

Let us now consider *bpark* and *barm* for a moment. *Bpark* specifies completely the way the arm is to be parked. It specifies the center of the hand by giving the Cartesian coordinates, and in addition it indicates that the hand is pointing downwards. Since there are six joints, specifying only the cart esian coordinates of the hand is insufficient since it is possible to have an infinite number of different hand orientations with the center of the finger tips in the same position.

Barm is the name of a coordinate system whose origin lies centrally between the fingers of the hand, and whose z-axis points in the same direction as the fingers, the y-axis passes through the centers of the fingers, and the x-axis is determined from these two axes by use of the right hand rule. The value of *barm* depends on the position and orientation of the hand, and consists of a vector which defines the position of the center of the hand in the world coordinate system, and a rot which defines how the arm coordinate system is rotated in terms of the coordinate system of the station. *Station* is the frame which is the reference coordinate system, and the vector part is set at (0,0,0). Our station coordinate system has the z-axis pointing upwards, the y-axis horizontal and parallel to the short side of the table and pointing towards the window (i.e in a direction pointing from the pedestal of the yellow arm to the pedestal of the blue arm). The x-axis is horizontal and parallel to the long side of the work table, and points towards the far wall,

In the park position the hand points downward with the center of the hand at coordinates (43.53, 56.86, 9.96) * inches. The coordinate system is rot at ed 180 degrees about the y-axis. Thus the value of *bpark* is as follows:

```
FRAME( ROT(YHAT,180*degrees),VECTOR(43.53, 56.86, 9.96)*inches)
```

The instruction *MOVE barm TO bpark* has the effect of moving the coordinate system whose name is *barm* to the new position and orientation described by *bpark*.

3.3 Using the fingers: *OPEN, CLOSE & CENTER*

Our manipulator end effector (hand) consists of two fingers which can move together or apart when instructed to do so by the *OPEN* or *CLOSE* command, which specifies the width to which the hand opening must go. An example of this particular instruct ion is

```
OPEN BHAND TO 2.5*inches
```

The general form of the instruction is

```
OPEN <hand> TO <scalar_exp>
CLOSE <hand> TO <scalar_exp>
```

where <hand> is either of the reserved words *bhand* or *yhand*, and <scalar_exp> consists of a scalar expression of dimension distance, i.e. its units should ultimately be reducible to inches or cm or some such unit of measure of distance.

The *OPEN* or *CLOSE* instruction moves both fingers simultaneously at the same speed. The *OPEN* command will open the hand to the desired size. The *CLOSE* instruction will keep on moving the finger until the touch sensors trigger, and signal an error if the hand opening is smaller than the desired opening. (The *CLOSE* instruction will be implemented in the near future.) If there is a heavy object between the fingers, the fingers or motors might get damaged, while a light object may get moved by the fingers. The *CENTER* command prevents these undesirable results by causing the fingers to move toward each other slowly until one of the touch sensors triggers to let the system know that contact has been made with the object. At this point the whole arm will shift to maintain the position of the finger which is in contact with the object, and the cycle of moving fingers and arm will continue until both touch sensors trigger. When this happens, the new position of the arm can be read to determine the position of the object. Note that the *CENTER* command does not "center" the object between the fingers, but rather ensures that the hand grasps the object without moving the object. The *OPEN* and *CLOSE* commands are used when the position of the object to be grasped is known precisely or when the object is to be moved to a precise spot. The *CENTER* command takes an arm as its argument as follows.

```
CENTER <arm>
```

The use of these statements will be illustrated in the following example used to grab a 2-inch cube, move it over 10 inches in the X direction, and then release it.

```
BEGIN
   FRAME cube, now-place;
   cube ←FRAME(ROT(XHAT,180*deg),VECTOR(20,30,1)*inch);
                         { defines position of cube center }
   now-place ←cube+10*xhat*inches;
   MOVE barm TO bpark WITH DURATION = 4*seconds;
   OPEN bhand TO 3*inches;
            {insure that we get barm and bhand to known positions }
   MOVE barm TO cube;          { get arm to planned location of cube }
   CENTER barm;                { grasp cube without moving it }
   MOVE barm TO new-place;     { put the cube where we want it }
   OPEN bhand TO 3.0 inches;   { open the hand, releasing the block }
   MOVE barm TO bpark;         { all done, park the arm }
END
```

3.4 Intermediate points - *VIA*, *APPROACH* and *DEPARTURE*

Many objects have shapes which necessitate care as the arm approaches or departs from them. The motion clause *WITH APPROACH = appr* will cause the arm to approach its destination after having passed through the point determined by vector *appr* in the coordinate *system* of the destination. In stat ion coordinates this point would be *dest + appr WRT dest.* The motion clause *WITH DEPARTURE = depr* similarly specifies a departure point. Section 4.4.3 indicates the effect of *appr* or *depr* taking on non-vector values.

If no approach point is given, a default approach of 3 inches along the Z axis of the station will be used. If no departure point is specified, the approach point from the last motion, if any, will be used. The word deproach (which is an abbreviation for departure and approach) has been coined to specify the general approach or departure point. Approach points relate to the destination of the current move command, while departure points relate to the starting posit ion of the arm for the current command. To move the arm directly from the frame position at the beginning of the motion, the clause *WITH DEPARTURE = NILDEPROACH* should be used. To move the arm directly towards the desired frame position indicated in the current statement, the clause *WITH APPROACH = NILDEPROACH* should be used.

If the destination is a frame constant or expression then *NILDEPROACH will* be the default approach point.

The predeclared macro *DIRECTLY* will accomplish the same purpose as the two clauses

WITH APPROACH = NILDEPROACH
WITH DEPARTURE = NILDEPROACH

The *APPROACH* and *DEPARTURE* clauses allow the user to specify at most a three segment motion - from the current position to the departure point, from the departure point to the approach point, and from the approach point to the destination. Usually t hese intermediate points are in terms of the coordinate system of either the current position or the destination.

Sometimes it is necessary to move an object through additional locations in space, or to have more than the three segment motions described above. Examples are cases where objects in the way of the moving manipulator have to be avoided, or the arm has to pass through an opening. In such situations the *VIA* clause maybbe used to specify the frames through which the arm must pass. The *VIA* clause should generally be associated with points which have values determinable at the planning stage. Points which can be determined only at runtime cause problems for the trajectory calculator.

In this example, the arm picks up a brick on the ground and places it on the floor of the oven, which is at the same level as the ground, but the arm has to pass through the oven door which is above ground level.

```
BEGIN "Put brick into oven"
    FRAME brick, ovon, oven-door;
    brick←FRAME(ROT(yhat,90*degrees) ,VECTOR(10,30,3)*inches);
                        {define initial position of brick }
    oven←FRAME(ROT(yhat,90*degrees),VECTOR(10,40,3)*inches);
                        {define final position of brick }
    oven_door←FRAME(ROT(yhat,90*degrees),VECTOR(15,40,4)*inches);
                        { dafino position of oven door }

    MOVE barm TO bpark WITH DURATION = 4*seconds;
    OPEN bhand TO 3*inches;
            { make sure arm and hand in known position }
    MOVE barm TO brick
        WITH APPROACH = 3*zhat*inches;
            { go for brick with hand in horizontal position,
              note that brick z-axis is parallel to station x-axis)
    CLOSE bhnnd TO 1.7*inches;
            { grasp the brick }
    MOVE barm TO oven VIA oven-door
        WITH DEPARTURE = -3*xhat*inches;
            { move brick into oven through oven door after lifting vertically }
    OPEN bhand TO 3.0*inches;
            {release the brick }
    MOVE barm TO bpark VIA oven-door;
            { go park the arm }
END
```

3.5 Modelling objects - affixment & indirect moves

Since assembly often involves attaching one object to anot her, AL has an automatic mechanism to keep track of the location of a subsidiary piece of the assembly as the main assembly is moved; the mechanism is called affixment. For example, there might be a frame called *pump* and another called *pump_base*. At some stage in the assembly, the *pump* is bolted to *pump-base*. At this time it is appropriate to execute the statement

```
AFFIX pump JO pump-base
```

This statement informs AL that motions of the *pump_base* are to affect the iocat ion of *pump*. Note that the *AFFIX* statement does not call any routines to generate the code to actually perform the bolting operation. Jhe statement merely informs AL that at this stage in the program execution, pump is to be considered affixed to *pump_base*.

The particular case in which object frames are attached to the arm frame is of special import ance. Once *pump* is affixed to *barm*, for instance, the user can forget about the arm, and just concentrate on where and how *pump* has to move; AL will take care of how to move the arm to achieve the desired result. This is an indirect move where the user need not specify arm motion,

When affixing frames to one another, the user must specify the relative transformation between the frames, and whether the affixment is rigid or nonrigid. The relative transformation can be specified within the affixment statement, or if the positions of the two frames are already defined, just stating that they are to be affixed will automatically compute the necessary trans.

The form of the affixment statement is as follows:

```
part ← <frame exp>;
fixture ← <frame exp>;
AFFIX part JO fixture NONRJGIDLY;
```

or alternately,

```
AFFIX pump JO pump-base AT <transexp> RIGIDLY;
```

RIGIDLY implies that the affixment is symmetric, so that changes in value of one frame imply changes in the other, A *RIGID* affixment is normally used when the objects are physically joined together rigidly, e.g. the pump being bolted to the *pump_base* or an arm grasping an object. In the above example, movement of pump will affect *pump_base*, and movement of *pump-base* will affect pump.

A *NONRIGID* affixment is used when one object is resting on another: e.g. *part* resting in *fixture*; *part* moves with the *fixture*, but if only *part* is moved, *fixture* stays put.

A frame could be affixed to more than one frame, and affixment s may be chained together. The affixment relationship can be broken by means of the UNFIX statement as follows:

UNFIX pump FROM barm;

Ail the frames rooted in *pump* (e.g. *pump_base*) will remain rooted in *pump*, and will no longer be affected by *barm* or its motion.

The following examples illustrate the stacking of one block on top of another with and without the use of affixment to illustrate its usage and convenience during programming.

```
BEGIN "block stacking without affixmont"

  FRAME blkl , bikl _grasp,blkl_top,blk2,blk2_grasp, finplace;
  DISTANCE SCALAR grasphoight, blkl length, blk2length, bikl width,
                  blk2width,blkl height;
  ROT stand;

  stand ←ROT(XHAT,180.*degrees);

  blkl width ←1.5*inches;        blk2widt h ←1.5*inches;
  blkl length ←2.4*inches;       blk2longth ← 2.4*inches;
  blkl height ←2*inches;         grasphoight ←0.75*inches;
                           { dofino dimensions of the blocks }

  blkl ← FRAME(nilrot,VECTOR(10,30,0)*inches);
  blk2 ←FRAME(nilrot,VECTOR(6,30,0)*inchos);
                           { ciefine bottom corner of blocks }
  finplace ← FRAME(nilrot,VECTOR(8,40,0)*inches);
                           {define final position of bottom of block 1 }

  blkl_grasp ← FRAME(stand,VECTOR(blkl length/2,blkl width/2,graspheight));
                           { define grasping position of block 1 }
  blkl,top ←FRAME(nilrot,VECTOR(0,0,blkl height));
                           { define position of top of block 1 }
  blk2_grasp ← FRAME(stand,VECTOR(blk2length/2,blk2width/2,graspheight));
                           { define grasping position of block 2 }

  MOVE barm TO bpark WITH DURATION=3*seconds;
  OPEN bhand TO 3.6*inches;

  MOVE barm TO blkl*blkl_grasp WITH APPROACH =3*zhat*inches;
                                     { arm moves to grasping position of blkl }
  CENTER barm;                       { hand grasps blkl}
  MOVE barm TO finplace*blkl_grasp WITH APPROACH =3*zhat*inches;
                                     { arm moves so that blkl is in final place }
  OPEN bhand TO 3.6*inches;          { hand opens to release blkl}
  MOVE bat-m TO blk2*blk2_grasp WITH APPROACH =3*zhat*inches;
                                    { arm moves to grasping position of blk2}
  CENTER barm;                      { hand grasps blk2 }
  MOVE barm TO finplace*blk1_top*blk2_grasp WITH APPROACH =3*zhat*inches;
                                    { arm moves to put blk2 on top of blk1}
  OPEN bhand TO3.6*inches;          { hand opens to release blk2 }

  MOVE barm TO bpark; PRINT ("all done");

END "block stacking without affixment";
```

Note that for each motion the destination is an expression consisting of a local coordinate system and a point in that system (e.g. *blk1*blk1_grasp*). Another way to write the same program is as follows, where AL automatically takes care of the bookkeeping of which coordinate system to use. The same number of declarations are still needed, but now the motion statements are clearer. Note that because the destination of each motion is no longer an expression AL will automatically use the standard approach.

```
BEGIN "block stacking using affixment"

  FRAME blk1 , blkl _grasp,blk1_top,blk2,blk2_grasp, finplace;
  DISTANCE SCALAR graspheight, blk1 length, blk2length,blk1 width,
          blk2width,blk1 height;
  ROT stand;

  stand ←ROT(XHAT,180.*degrees);

  blk1 width ← 1.5*inches;         blk2width ← 1.5*inches;
  blk1 length ←2.4*inches;         blk2length ← 2.4*inches;
  blk1 height ←2*inches;           graspheight ←0.75*inches;

  blk1 ← FRAME(nilrot,VECTOR( 10,30,0)*inches);
  blk2 ←FRAME(nilrot,VECTOR(6,30,0)*inches);
  finplace ← FRAME(nilrot,VECTOR(8,40,0)*inches);

  AFFIX blk1_grasp TO blk1 at
       TRANS(stand,VECTOR(blk1length/2,blk1width/2,graspheight)) RIGIDLY;
  AFFIX blk1_top T O blk1 at
       TRANS(nilrot,VECTOR(0,0,blk1 height)) RIGIDLY;
     { top and grasping position of block1 are defined with respect to bottom }
  AFFIX blk2_grasp TO blk2 at
       TRANS(stand,VECTOR(blk2length/2,blk2width/2,graspheight)) RIGIDLY;
     { grasping position of block2 defined with respect to bottom }

  MOVE barm TO bpark WITH DURATION =3*seconds;
  OPEN bhand TO 3.6*inches;
          { normalize arm position };

  MOVE barm TO blk1_grasp;     { arm moves over the grasping position of blk1}
  CENTER barm;                 { hand closes over blk1}
  AFFIX blkl to barm RIGIDLY;  {blk1 and all its parts are attached to arm }
  MOVE blkl TO finplace;       { note that blkl is moved, not barm }
  OPEN bhand TO 3.6*inches;    { this physically releases the block }
  UNFIX blk1 from barm;        { blkl is released from the arm in the world model }

  MOVE barm TO blk2_grasp;
  CENTER barm;
  AFFIX blk2 to barm RIGIDLY;
  MOVE blk2 TO blkl,top;       { move bottom of blk2 to the top of blkl}
  OPEN bhand TO 3.6*inches;
  UNFIX blk2 from barm;

  MOVE barm TO bpark;  PRINT ("all done");

END "block stacking using affixment";
```

3.6 Sensing forces - simple condition monitors

When we want to use threshold values of sensory information to perform certain actions, we make use of condition monitor clauses. The syntax is as follows:

```
ON <condition> DO <action>
```

A simple example would be to rotate the wrist of the arm (assumed vertical) and stop when a torque of 50 ounce-inches is encountered - perhaps that indicates that we have tightened something to the required torque. An example of such a statement would then be

```
MOVE barm TO barm*FRAME(ROT(zhat,90*degrees),nilvect*inches)
      ON TORQUE(zhat) ≥ 50 *ounces*inches DO STOP barm;
```

The effect of this statement is obvious; the *STOP* command stops the motion of the arm immediately after the force is encountered. Note the specification of the direct ion of the detected torque, *zhat*, and the threshold amount (50 ounce-inches),

Assume we want to find the height of an object and that the object is 'expected to be in a given location, and that its height is expected to be between 2 and 12 inches.

```
BEGIN
  FRAME object;
  DISTANCE SCALAR height;

  MOVE barm TO bpark WITH DURATION=3*seconds;
  CLOSE bhand TO 0*inches;  { bring fingers together }

  MOVE barm TO object +14*zhat*inches;  { arm is vertically above the object }

  MOVE barm TO ⊗-13*zhat*inches       { symbol ⊗ here means current position of barm }
     WITH DURATION =10*seconds
     ON FORCE(ZHAT)≥10*ounces DO STOP;
          { try to move arm down 13 inches slowly and stop when a force is
            encountered; i.e. contact is made }

  height ←POS(barm).zhat - 0.3*inches;
```

```
                { take the z-component of the arm's current location and subtract the
                distance between the center and edge of the fingers to give the actual
                height of the object }

        PRINT("HEIGHT OF OBJECT IS ", height, " INCHES");
    END;
```

3.7 Applying forces & compliance

In addition to detecting forces, AL allows specified forces to be applied. Since forces have both direction and magnitude, the applied force must have both specified, either in terms of the resultant magnitude and direction, or in terms of orthogonal components along the principal axes of a given coordinate system. Applying a force of magnitude zero means that the arm will be compliant, i.e. move away from any external force in that direction. In the following example, the arm is compliant to forces in the x and y directions (Le., it tends to move away from any external forces in those directions), while it applies a downward force of 10 ounces in the z direction. The *FORCE_FRAME* clause indicates the coordinate system in which the force components are specified, and is needed whenever two or more force components (which must be orthogonal and along the principal axes) are used. The above is also applicable to torques. In the example below, this coordinate system is in world (fixed) coordinates and has the station orientation. *FORCE_FRA* ME is described in more detail in Chapter 4.

```
BEGIN "insert peg into hole"
    FRAME peg-bottom, peg-grasp, hole-bottom, hole-top;

    MOVE barm TO bpark WITH DURATION=3*seconds;
    OPEN bhand to 3*inches;      { normal initialization }

    peg-bottom ←FRAME(nilrot,VECTOR(20,30,0)*inches);
    hole-bottom ←FRAME(nilrot,VECTOR(25,35,0)*inches);

    AFFIX peg_grasp TO peg_bottom RIGIDLY
        AT TRANS(ROT(xhat,180*degrees),3*zhat*inches);
    AFFIX hole-top TO hole-bottom RIGIDLY
        AT TRANS(nilrot,3*zhat*inches);

    MOVE barm TO peg-grasp;
    CENTER barm;                 { get peg }
    AFFIX peg_grasp TO barm RIGIDLY;
    MOVE peg-bottom TO hole-top;
    MOVE peg_bottom TO hole-bottom DIRECTLY      [prevent arm lifting and dropping)
        WITH FORCE-FRAME = station IN WORLD
        WITH FORCE(zhat) = -1 0*ounces       {force components in station coordinates)
        WITH FORCE(xhat)=0*ounces
        WITH FORCE(yhat) = 0*ounces
        SLOWLY;      { SLOWLY is a macro which slows movements by 3 times
                         (c.f. section 3.14.4)}
END "insert peg into hole";
```

When force is applied, there should be a resisting force, otherwise the arm will accelerate in the direction specified because of Newton's Second Law.

However, compliant motions (forces in certain directions equal zero) currently cause spont aneous motion even if there is nothing touching the arm because noise amplification and imprecise modeiling of the arm loads and geometry may cause an initial velocity which the arm tries to maintain because of Newton's First Law. It is expected that this problem may be alleviated by the use of damping. Finkel, in "Constructing and Debugging Manipulator Programs", discusses some of the problems associated with compliant motion specifications.

3.8 Control structures: IF, FOR & *WHILE* statements

AL has many of the traditional ALGOL control structures, including condi tionals and loops. There are no jumps in AL, because they confuse the flow analysis needed for rnaintaining planning values. In this sect ion we shall describe the *IF*, FOR and *WHILE* statements.

The *IF* statement has the form:

```
IF <condition>
   THEN <statement>
   ELSE <statement>
```

The *ELSE* part is optional. The condition is some boolean expression involving one of the operators $<, >, \leq, \geq, =$, and $\neq$. Boolean expressions can be built up out of relational operators, the logical connectives $\wedge$ (*AND*), $\vee$ (OR), $\neg$ (*NOT*), $\otimes$ (*XOR*, exclusive or), $\equiv$ (*EQV*, the logical equivalence) or the logical constants *TRUE* or *FALSE*. The condition may also be some arithmetic scalar expression. If the condition is true (non-zero) the statement following the *THEN* is executed. Otherwise the statement following the ELSE, if present, will be executed.

The *FOR* loop has the form:

```
FOR <svar>←<sexpr> STEP <sexpr> UNTIL <sexpr> DO <statement>
```

where *<svar>* stands for "scalar variable" and *<sexpr>* stands for "scalar expression of same dimension". The initial value of the variable is the value of the first expression; every time tho statement is executed, its value is incremented by the value of the second expression, and the process repeats until the value exceeds that of the third expression. if the step size is negative, the right things happen. A test is made before the first iteration, so it is possible that the loop will not get executed at all.

The *WHILE* loop is as follows:

```
WHILE <condition> DO <statement>
```

where *<condition>* is the same as above. The condition is checked and if it is true the statement is executed. The process is repeated until the condition becomes false.

The following example illustrates the use of the *IF*, *FOR* and *WHILE* statements in a program where the arm picks up castings from one place, puts the good ones on a pallet in 6 rows of 4 and discards the defective ones. The castings come in batches of 50, but it is not known ahead of time how many batches there will be.

```
BEGIN "sort castings"
    FRAME pickup, garbage-bin, pallet;
    SCALAR pallet_row,pallet_column, good, bad;
    DISTANCE SCALAR packing-distance;
    SCALAR ok, more-batches, casting-number;
    packing_distance←4*inches;

    MOVE barm TO pickup WITH DURATION = 3*seconds;
    OPEN bhand TO 3*inches;

    pallet-row+ 1; pallet_column←0; good←0; bad+-0;
    casting←pickup;

    MOVE barm TO pickup DIRECTLY;
    CENTER barm;
    IF (bhand <1.5*inches) THEN more_batches←FALSE ELSE more_batches←TRUE;

    WHILE more-batches DO
        BEGIN "sort 50 castings"

        FOR casting-number+ 1 STEP 1 UNTIL 50 DO
            BEGIN "sort casting in hand"
            ok ← FALSE;
            AFFIX casting TO barm RIGIDLY;
            MOVE casting TO pickup • 3*zhat*inches
                ON FORCE(zhat)≥20*ounces DO ok←TRUE; {see if it weighs enough }

            IF ok THEN
                BEGIN "good casting"
                good+-good+ 1;
                IF pallet_column=4
                    THEN BEGIN pallet_column←0;pallet_row←pallet_row • 1; END
                    ELSE pallet_column←pallet_column+1;
                MOVE casting TO pallet +
                      VECTOR( pallet-column*packing-distance,
                              pallet_row*packing_distance,0*inches)
                      WITH APPROACH =3*zhat*inches;
                UNFIX casting FROM barm;
                OPEN bhand TO 3*inches;
                IF (pallet_column=4) AND (pallet_row=6)
                    THEN BEGIN "pallet full"
                    pallet_column←0;pallet_row← 1;
                    { code to remove this pallet and get new pallet }
                    END "pallet full";
                MOVE barm TO pickup;
                END "good casting"
            ELSE
```

```
                    BEGIN "defective casting"
                    bad←bad+1;
                    MOVE casting TO garbage-bin DIRECTLY;
                    OPEN bhand TO 3*inches;
                    UNFIX casting FROM barm;
                    MOVE barm TO pickup;
                    END "defective casting";

                casting←pickup;
                CENTER barm;
                END "sort casting in hand";

            IF (bhand <1.5*inches) THEN more-batches ← FALSE;
            END "sort 50 castings";

        MOVE barm TO bpark;

        PRINT("THERE WERE ", good," GOOD CASTINGS AND ", bad," DEFECTIVE CASTINGS");

    END "sort castings";
```

3.9 Control structures (cont): *CASE* & *UNTIL* statements

Two of the other traditional ALGOL control structures in AL are the *CASE* and *UNTIL* statements.

The *CASE* statement comes in several forms, The regular *CASE* statement has the form:

```
CASE <index> OF
    BEGIN
    <statement 0>;
    <statement 1>;
    <statement 2>;

    <statement n>
    END
```

The. scalar index expression is evaluated and depending on the integer part of its value one of the following statements is executed. If the index is zero then statement 0 is chosen, if the index is one then statement 1 is chosen, and so on up till n. If the index is negative, or greater than the number of statements, an error is reported. Any of the statements may be null, e.g. "<statement 1>;;<statement 3>", in which case if the index were two nothing would be done.

There is also a numbered version of the *CASE* statement:

```
CASE <index> OF
    BEGIN
```

```
[CO] <st at ement>;
[C1] <statement>;
[C2]<st at ement>;


[Cn] <statement>;
ELSE <st at ement>
    END
```

where each statement has one or more non-negative scalar constants labelling it. Again, the index expression is evaluated and if its integer part is the same as one of the Ci's then the st atement with that label is executed, Otherwise, if an *ELSE* is present, the statement it labels is executed. If no *ELSE* is present, an error occurs if the integer part of the index is negative or greater than the largest Ci, otherwise nothing is done. Note that the *ELSE* statement may appear anywhere in the list of statements; it need not be at the end.

Here is an example using the numbered *CASE* statement to select the appropriate action to perform when given one of several possible parts.

```
BEGIN
  SCALAR  part-number;
  FRAME pick_up,base,base_grasp,cover,cover_grasp,side,side_grasp,...;

 (Initialization code including the following macro definitions:
  DEFINE base,num =...;
  DEFINE cover-num = ...;
  DEFINE side-num = ...;
  which will be used for clarity in a numbered case statement.}

 (Now go get the part at pick-up and do whatever is appropriate with it.)

  PRINT("Entor tho part's number: ");
  part-number ← INSCALAR;    (INSCALAR reads in a scalar from the console keyboard }
      (Have the user type in the part's number. In the future this might
       be done automatically using vision,}
  CASE part-number OF
      BEGIN
      [base_num] BEGIN {Code to handle base.)
                     base ← pick-up;
                     MOVE barm TO base-grasp;
                     CENTER barm; (Grab it}
                     AFFIX base TO barm;
                     {Rest of code for base.}
                 END;

      [cover-t-turn)  BEGIN
                     (Code to handle cover.)
                 END;

      (Repeat for other known parts: side,etc.}

      ELSE        BEGIN
                     PRINT("Unknown part number",crlf);
```

```
                        (Code to recover from error)
                      END
          END;

          {Rest of program. Note that each of the statements in the above CASE statement
    should lcavo the arm in the same position.  If not then a plan-time assign (c.f. section
    4.5.1) will be required so the world modeller knows the arm's position when the CASE
    statement finishes. (c.f. sections 4.4.1 & 4.6) }
    END;
```

The *UNTIL* statement is as follows:

```
                    DO <statement> UNTIL <condition>
```

where the statement is repeatedly executed until the condition becomes true. This is similar to the *WHILE* statement described in the previous section, with the exception that the *WHILE* loops while the condition is true, whereas the *UNTIL* loops until the condition becomes true, Note that the body of an *UNTIL* loop is always executed at least once.

As an example of the use of the *UNTIL* statement, here is a program excerpt that gets a good casting, discarding any bad ones it finds in the process. It is similar to the exarnple in the previous section.

```
BEGIN
  SCALAR success;
  (Initialization code}

  success ← false;
  casting←pickup;
  MOVE barm TO casting-grasp;

  DO BEGIN (Try to get a good casting}

      CENTER barm;
      AFFIX casting TO barm RIGIDLY;
      MOVE casting TO pickup +3*zhat*inches       {See if it weighs enough)
         ON FORCE ≥20*ounces ALONG zhat OF station DO success ← true;

      IF ¬success THEN (Get rid of defective casting)
         BEGIN
         MOVE casting TO garbage-bin DIRECTLY;
         OPEN bhand TO 3*inches;
         UNFIX casting FROM barm;
         casting+pickup;
         MOVE barm TO casting-grasp
         END

      END UNTIL success;

   barm ←← pick-up +3*zhat*inches;
            (Plan-time assign so the world modeller will be happy}

  {Code for rest of program)
END;
```

3.10 Simultaneous motion: *COBEGIN-COEND, SIGNAL-WAIT*

So far we have considered single arm moves. To perform simultaneous movements of arms, two new concepts have to be introduced. The *COBEGIN-COEND* block has the same effect as the *BEGIN-END* block, except that statements within the block are executed simultaneously.

Thus the following will park both arms at the same time.

```
COBEGIN
   MOVE barm TO bpark;
   MOVE yarm TO ypark;
COEND;
```

Simple synchronization is possible within the context of sirnultaneous execution. This is achieved by means of explicit events and the *SIGNAL* and *WAIT* statements. Every different event that the user wishes to use should be declared in a declaration st at ement as follows;

```
EVENT e 1,e2,e3
```

The *EVENT* is distinct from algebraic data types (e.g. scalars) and cannot be assigned a particular value by the user in his program by means of the regular assignment statement. With each event is associated a count of how many times it has been signalled. Initially, the count is zero, that is, no signals have appeared, and no processes are waiting. The statement

```
SIGNAL e l
```

increments the count associated with event *el*, and if the resulting count is zero or negative, one of those processes waiting for *el* is released from its wait and readied for execution. The statement

```
WAIT e1
```

decrements the count associated with event *el*, and if the resulting count is negative, the process issuing the *WAIT* is blocked from continuing until a signal comes along. If the count is zero or positive, there is no waiting.

The following example is used to show the use of the *SIGNAL* and *WAIT* commands, although it rnay be done without these constructs. The blue arm picks up an object and moves to a passing location, where it makes sure that the yellow arm has grasped it before releasing it.

```
BEGIN
   EVENT passed, caught, ready-pass;
   FRAME steel_beam, pass, catch;

   COBEGIN
      BEGIN "blue"
      MOVE barm TO steel-beam;
      CENTER barm;
      AFFIX steel-beam TO barm;       { barm gets steel beam }
      MOVE stool-beam TO pass;        { takes it to passing position }
      SIGNAL ready-pass;              { barm says it is ready }
      WAIT caught;                    { waits for yellow arm to catch)
      OPEN bhand TO 3.0*inches;       { when yellow arm ready releases beam)
      UNFIX steel-beam FROM barm;
      SIGNAL passed;                  { barm announces it has released beam }
      END "blue";

      BEGIN "yellow"
      OPEN yhand TO 3.0*inches;       { meanwhile yellow hand is opened }
      MOVE yarm TO catch;             { yellow arm goes to catching position }
      WAIT ready-pass;                { yarm waits till there is something to grab)
      CENTER yarm;                    { grasps it)
      SIGNAL caught;                  { yarm announces it caught it }
      WAIT passod;                    { waits for blue arm to release it }
      MOVE yarm TO pallet;
      END "yellow";
   COEND;
END;
```

A second example illustrates the use of *SIGNAL* and *WAIT* in resource sharing. The example in the last section where castings are sorted will be used but assume that the two arms are doing similar jobs, and that a single overhead crane is used to take away the full pallets and bring in empty pallets. Blue and yellow pallets are used to correspond to the appropriate arms. The code for the program will be similar to the previous section, except that the section which states *{code to remove this pallet and get new pallet}*, in the block labeled "*pallet full*", will use *SIGNAL* and *WAIT* to ensure that the crane is not asked to go to two locations at the same time, and that it is asked to go to a location only when it is needed.

```
BEGIN
   EVENT blue_pallet_full, blue-pallet-empty;
   EVENT yellow_pallet_full, yellow-pallet-empty;
   EVENT crane-free;
   SCALAR more_blue_pallets,more_yellow_pallets;

   more_blue_pallets←TRUE;     more_yellow_pallets←TRUE;
   SIGNAL crane-free;

   COBEGIN
      BEGIN "load blue pallets"
         BEGIN "sort castings"  {code from section 3.8)
```

```
                IF (pallet_column=4) AND (pallet_row=6)
                THEN BEGIN "pallet full"
                     pallet_column←0; pallet_row← 1;
                     SIGNAL blue_pallet_full;
                     WAIT  blue-pallet-empty;
                     END "pallet full";



            END "sort castings";
            SIGNAL blue_pallet_full;        (to get last pallet out of the way)
            WAIT blue-pallet-empty;
            more_blue_pallets←FALSE;        {to stop crane waiting for blue pallet,
                                            otherwise crane program will get stuck in
                                            "change blue pallet" block.)
        END;

        BEGIN "load yellow pallets"
            BEGIN "sort castings"  (similar to blue pallets except use yellow
                                   arm and yellow pallet)



              IF (pallet_column=4) A N D (pallet_row=6)
                THEN BEGIN "pallet full"
                     pallet_column←0;pallet_row← 1 ;
                     SIGNAL yellow_pallet_full;
                     WAIT yellow_pallet_empty;
                     END "pallet full";



            END "sort castings";
            SIGNAL yellow_pallet_full;
            WAIT yellow_pallet_empty;
            more_yellow_pallets←FALSE;
        END;

        WHILE more_blue_pallets
        DO BEGIN "change blue pallet"
                WAIT blue_pallet_full;
                WAIT crane-free;                { wait for crane to be free }
                      (code to use crane to change blue pallet}
                SIGNAL blue_pallet_empty;
                SIGNAL  crane-free;
           END;

        WHILE more_yellow_pallets
        DO BEGIN "change yellow pallet"
                WAIT yellow_pallet_full;
                WAIT crane-free;                { wait for crane to be free }
                      {code to use crane to change yellow pallet}
                SIGNAL  yellow-pallet-empty;
                SIGNAL  crane-free;
           END;

    COEND;
END;
```

3.1 1 Arrays

Sometimes we would like a variable to refer to more than *one* value. As an example consider a base plate with three screw holes in it. During the assembly, code to insert a screw into each hole will be writ ten. Rat her than repeatedly writing the same code for each screw hole, it would be preferable to write it once and somehow use a *FOR* loop to repeat it for all the holes. An array will allow us to do this.

An array is a variable that can have multiple values. In the above example we had three frames; first-hole, second-hole and third-hole. We can define a frame array: hole[1:3] which allows us to reference the three screw holes as: hole[1], hole[2] and hole[3]. More formally an array definition is of the form:

```
<type> ARRAY <name1>[bounds],<name2>[bounds]
```

where type specifies the array's data type, and bounds indicates the size of the array and how the elements of it are referenced. Our example above used a one dimensional array. An example of a two dimensional array is:

```
SCALAR ARRAY foo[1:3,1:4]
```

which would look like;

```
foo[1,1]  foo[1,2]  foo[1,3]  foo[1,4]
foo[2,1]  foo[2,2]  foo[2,3]  foo[2,4]
foo[3,1]  foo[3,2]  foo[3,3]  foo[3,4]
```

There is no upper limit on the number of dimensions an array rnay have. The array bound pairs may be either scalar constants, variables or expressions. The bounds -may have positive or negative values, as long as the lower bound is smaller than the upper bound. For example:

```
VECTOR ARRAY u[-3:3],v[n:n+5],w[0:3,1:m]
```

where n and m are scalar variables. Space is allocated for arrays upon entry of the block in which they are defined, so the sizes of v and w will depend on the values of n and m when the definition occurs,

Arrays are used in programs just like regular variables. For example:

```
FOR i ← 1 STEP 1 UNTIL 4 DO foo[ 1 ,i] ← foo[ 2,i]*foo[3,i]
```

At runtime a check is made that each subscript falls within the lower and upper bounds given for the dimension it specifies. Subscripts out side the bounds cause an error message to be printed. Only the integer part of the subscript is used.

Here is an example to do the screw insertion task mentioned at the beginning of this section.

```
     BEGIN
        FRAME ARRAY hole[1:3];
        FRAME  base-plate;
        SCALAR i;

       (Initialization and start of the program including definition of the locations of
        the base-plate and the screw holes:
        base-plate ←FRAME(....);
        AFFIX hole[1] TO base-plate RIGIDLY AT TRANS(....);
        AFFIX hole[2] TO base-plate RIGIDLY AT TRANS(....);
        AFFIX hole[3] TO base-plate RIGIDLY AT TRANS(....);
        Screws will bo defined with the z-axis pointing downward.
        Code to get the screw driver into the hand is also included. }

       (Now insert the three screws}

        FOR i ← 1 STEP 1 UNTIL 3 DO
          BEGIN
            screw← screw,disponsor;          {Define location of new screw}
            MOVE driver-tip TO screw;        {Get a screw - not really this easy)
            AFFIX screw TO driver;
            MOVE screw-tip TO hole[i];       (Screw is just above screw hole}

            COBEGIN
               MOVE screw TO ⊗- 0.75 *zhat* inches          {Push down with arm)
                    WITH FORCE = 20 * ounces ALONG zhat OF screw
                    WITH DURATION = 2.5 seconds;
               OPERATE driver                    (Drive in the screw}
                    WITH VELOCITY = 200 * rpm
                    WITH DURATION = 3 * seconds;
            COEND;

            UNFIX screw FROM driver              {Release the screw)
          END;
     END
```

Note that the “driver” used above is not available currently.

3.12 Procedures

There are times when we wish to do the same operation at several places in the program. Rather than place the entire sequence at each of these points it is often desirable to code it up once as the body of a procedure or subroutine, and at each point in the program where the operation is required have a call on the procedure. As an example during an assembly there may be a number of screws that need to be inserted. A procedure to do this insertion will be shown after the syntax for procedures has been explained.

Procedures are defined as follows:

```
<type> PROCEDURE <name> (parameter list);
    <statement>
```

where the statement is executed each time the procedure is called. A simple procedure to park the arm and open the fingers could be written as:

```
PROCEDURE park;
    BEGIN
    MOVE barm TO bpark WITH DURATION =3*sec;
    OPEN bhand TO 3 * inches;
    END;
```

Any time in the program the user wants to move the arm to the park position and open the hand all she need type is the statement:

```
park
```

which will call the procedure. Sometimes a procedure will be used to return a result needed for computation (i.e., the proceduce will be used as a function). This is done by use of the *RETURN* statement:

```
RETURN (value)
```

which returns value as the result of the procedure. For example a procedure to determine the height of the blue arm might be writ ten:

```
DISTANCE SCALAR PROCEDURE height,barm;
    RETURN(POS(barm) . zhat );
```

Any time the height of the blue arm is needed one would call the procedure. Note the declaration of the data type that the procedure returns. We can generalize this procedure so that for any frame it returns the height of the frame. To do this we introduce the use of parameters to pass a value to the procedure. The generalized procedure and a sample of it in use is as follows:

```
DISTANCE SCALAR PROCEDURE height (FRAME f);
    RETURN(POS(f) . zhat );

PRINT("The height of the pallet is:", height(pallet_top));
```

when the procedure is called the parameter *f* is bound to the value of pallet-top, and every reference to *f* in the body of the procedure will refer to pallet-top. Parameters can be passed by reference, which is the default for variables and arrays, or by value, the only way expressions are passed. If a variable is passed by reference then its value can be modified by the procedure. For example a procedure to refine the location of a frame by grasping it with the arm and then reading the position of the arm might be written:

```
PROCEDURE refine (REFERENCE FRAME obj);
      BEGIN
      OPEN bhand TO 3*inches;
      MOVE barm TO obj;
      CENTER barm; {This will sense the object's position)
      obj ← barm
      END;
```

When the procedure returns, the frame passed as its argument will have a new value.

A traditional example of a procedure used in most programming tutorials is the factorial function: fact(1) = 1, fact(2) = 2*1, fact(3) = 3*2*1, etc. Here are two ways of writing factorial in AL; the first is iterative, while the second is recursive (i.e. it calls itself).

```
SCALAR PROCEDURE ifact (SCALAR n);
      BEGIN
      SCALAR i, prod;
      prod ← 1;
      FOR i ← 2 STEP 1 UNTIL n DO prod ← prod * i;
      RETURN( prod );
      END;

SCALAR PROCEDURE rfact (SCALAR n);
      IF n >1 THEN RETURN( n *rfact(n-1) ) )
                   ELSE RETURN(1);
```

A procedure to do the screw insertion operation is as follows:

```
PROCEDURE insert-screw (FRAME hole-location);
  BEGIN
   screw←screw_dispenser;
   MOVE driver-tip TO screw;          (Get a screw - not really this easy)
   AFFIX screw TO driver;
   MOVE screw_tip TO hole_location;         (Screw is just above screw hole}

   COBEGIN
     MOVE screw TO ⊗- 0.75 * zhat * inches          (Push down with arm)
         WITH FORCE = 20 * ounces ALONG zhat OF screw
         WITH DURATION = 2.5 seconds;
     OPERATE driver                          {Drive in the screw}
         WITH VELOCITY = 200 * rpm
         WITH DURATION = 3 * seconds;
   COEND;

   UNFIX screw FROM driver              (Release the screw>
  END;
```

Now the loop to insert three screws in the example in the previous section would be:

```
FOR i ← 1 STEP 1 UNTIL 3 DO insert_screw(hole[i]);
```

It should be mentioned that procedures present certain difficulties to the world modeller in the AL compiler. Please refer to section 4.6 for a discussion of these problems and solutions to them.

3.13 Hints to the Programmer

3.13.1 Upward pointing grasping posit ions

The AL user will quickly realize that under normal usage, the frame *barm* usually has its Z axis pointing downwards in station coordinates. Since we are used to thinking in terms of an upward positive Z direction, it is sometimes convenient to define another frame affixed rigidly to *barm* but with the Z-axis pointing upwards, and the Y axis either parallel or anti-parallel to the station Y axis. With such a frame, the user can define grasping frames with the station orientation if the hand points downwards. The following statements will set up a frame called *bgrasp* to accomplish what we want.

```
FRAME bgrasp;
AFFIX bgrasp TO barm AT TRANS(ROT(xhat,180*deg),nilvect*inches) RIGIDLY;
```

3.13.2 Initialization and program end

Initialization of the arm and hand to known positions before starting is a good idea to ensure that the first movement from an unknown position does not result in the arm trying to move too fast,

The statements recommended are:

```
MOVE barm TO bpark WITH DURATION = 3*seconds;
OPEN bhand TO 3*inches;
```

It is good policy to park the arm at the end of the program by using:

```
MOVE barm TO bpark
```

The AL compiler will give a warning message if the arm is not parked upon program completion.

3.13.3 Slowing down movements

When trying out a program for the first time when it is not known how the arm will behave, the use of a *speed_factor* greater than unity will slow down all motions in the program (c.f. section 4.4.6 for details), The user should assign a value to *speed_factor* at the beginning of the program as follows:

```
speed-factor ← 2.0
```

For convenience, two predeclared macros *SLOW* and *CAUTIOUS* assigning values of 2.0 and 3.0 respectively to *speed_factor* may be used instead of the assignment statement described above.

4. THE AL LANGUAGE

AL is an ALGOL-like source language extended to handle the problems of manipulator control. This chapter describes the features of the AL language. It is presumed that the reader has read the previous chapter which introduces the AL language in a tutorial fashion.

4.1 Basic constructs

4.1.1 Programs

AL programs are organized in the traditional block structure of ALGOL. A program in AL consists of either a single statement or a block statement, which is a sequence of statements, separated by semicolons, and surrounded by the reserved words *BEGIN* and *END* (or *COBEGIN* and *COEND*). Blocks may be named by placing a string constant immediately after the *BEGIN* (or *COBEGIN*). This name will be checked against the string (if any) that follows the matching *END* (or *COEND*), and if the two strings do not match, an error will be reported.

BEGIN “block name” S; S; S; S END “block name”

4.1.2 Variables

A variable name is a string of alphanumeric characters and underscore, "_", starting with a letter. Variables must be declared before being used. AL follows normal variable scoping rules: variables may only be referenced within the block they are declared in, or in blocks nested within that block, The same variable name may be declared in several blocks, in which case any references to it refer to the innermost declaration enclosing the reference.

4.1.3 Comments

Comments are text inserted into the program to make it more readable. Comments can be written in two forms. The compiler will ignore all text between the reserved word *COMMENT* and the next semicolon encountered. Comments may also be enclosed by curly brackets "{}".

4.2 Data types and expressions

4.2.1 Algebraic data types: *SCALAR, VECTOR, ROT, FRAME, TRANS*

The basic data types in AL were chosen to facilitate working in the three dimensions of the real world. Scalars are -floating point numbers like reals in other computer languages. Vectors are 3-tuples specifying (X, Y, Z) values, which represent quantities like translations, velocities, and locations with respect to some coordinate system. Rot ations are 3x3 matrices representing either an orient at ion or a rotation about an axis. A rotation, or rot, is constructed from a vector, specifying the axis of rotation, and a scalar, giving the angle of rotation. Frames are used to represent local coordinate systems, They consist of a vector specifying the location of the origin, and a rotation specifying the orientation of the axes. Transes are used to transform frames and vectors from one coordinate system to another. Like frames they consist of a vector and a rotation.

4.2.2 Labels & Events

Labels and events are data types that are declared in the same manner as the algebraic data types. There are two kinds of labels: statement labels and condition monitor labels. Condition monitors are labelled for reference by the *ENABLE* and *DISABLE* statements (c.f. section 4.4.5.2). Statements are labelled for use in debugging. A label consists of an identifier followed by a colon. Currently labels must be declared before being used.

Events are used in conjunction with the *SIGNAL* and *WAIT* statements (c.f. section 4.5.4) used to synchronize parallel processes.

4.2.3 Arrays

Multi-dimensional arrays are available in AL. They may be of any algebraic data type or of type event. Array bounds may be scalar constants, variables, or expressions; they may be positive or negative integers. The only constraint is that the lower bound be smaller than the upper bound. At runtime a check is made that each subscript falls within the lower and upper bounds given for the dimension it specifies. Subscripts outside the bounds cause an error message to be printed.

Arrays are allocated upon entry of the block in which they are defined, and deallocated upon block exit,

4.2.4 Dimensions

AL allows physical dimensions to be associated with variables. The known dimensions are: *TIME, DISTANCE, ANGLE, FORCE, TORQUE, VELOCITY, ANGULAR_VELOCITY & DIMENSIONLESS.* New dimensions may be defined if desired by means of the *DIMENSION* statement.:

DIMENSION <new dimension> = <dimension expression>

where the operators defined in <dimension expression> are (,),*,/ and *INV*, which takes the inverse of its argument, e.g. INV(TIME) = 1 /TIME.

Dirnensioned quantities are just like regular ones, except that they are multiplied by the appropriate reserved word: *SEC, CM, DEG, GM, INCHES, OZ & LBS* (also *SECONDS, INCH, OUNCES, DEGREES & RADIANS).* For example:

```
VELOCITY VECTOR v;
v ← xhat * inches /sec
```

Other units rnay be defined using macros (cf. section 4.5.8), e.g.:

```
DEFINE feet = ⊂( 12 *inches)⊃
```

AL checks for consistent usage of dimensioned quantities: addition and subtraction, along with frame, trans and rot operations require exact dimension match, while scalar and vector multiplication and division produce a quantity of new di rnensi on,

4.2.5 Declarations

The declaration statement is used to define the data type and dimension of each variable used in a program. it has the form:

<dimension> <data type> <list of variables>

where *<dimension>* is one of the predefined dimensions in AL *(TIME, DISTANCE, ANGLE, FORCE, TORQUE, VELOCITY & ANGULAR_VELOCITY)*, or a user defined dimension. *<Datatype>* is one of the following: *SCALAR, VECTOR, ROT, FRAME, TRANS, EVENT & LABEL.* Only the algebraic data types: *SCALAR, VECTOR* and *TRANS* rnay have a dimension associated with them. Unless otherwise specified, scalars and vectors are considered dimensionless, while transes are considered to be of dimension distance (cf. section 3.1.1.5).

Array declarations are of the form;

<dimension> <data type> ARRAY <list of variables>

where each variable in the list consists of a variable name followed by a list of lower-upper bounds pairs enclosed in square brackets "[]", e.g. "name[L1:U1, L2:U2,...]".

4.2.6 Arithmetic expressions

Here is a summary of the arithmetic operators available. They are grouped by the data type of their resulting value. These abbreviations are used: 's' = scalar, 'v' = vector, 'r' = rotation, 'f' = frame, 't' = trans.

Scalar operators

s + s	scalar addition
s - s	scalar subtract ion
s * s	scalar multiplication
s / s	scalar division
s ↑ s	scalar raised to a scalar power
s MAX s	maximum
s MIN s	minimum
INT(s)	integer part of s
s DIV s	integer quotient after applying INT to each argument
s MOD s	integer remainder after applying INT to each argument
v . v	dot product of two vectors
|s|	absolute value of a scalar
|v|	magnitude of vector (vector norm)
|r|	extracts angle of rot ation
INSCALAR	reads a scalar from the console

Scalar functions

SQRT(s)	square root
SIN(s)	sine (all trigonometric functions are in degrees)
COS(s)	cosine
TAN(s)	tangent
ASIN(s)	arc-sine
ACOS(s)	arc-cosine
ATAN2(s,s)	arc-tangent of s/s
LOG(s)	natural logarithm
EXP(s)	e raised to the s power

Boolean operators

s <rel> s	returns true if relation is satisfied, else false possible relations are: $<, \leq, =, \geq, >, \neq$
S A S	logical and
s v s	logical or
s ⊗ s	logical exclusive or
s ≡ s	logical equivalence
¬ s	logical not
QUERY	reads a boolean from the console (c.f. section 4.5.7)

Vector operators

VECTOR{ s,s,s)	construct vector given (x,y,z) components
s * v	dilation of a vector
v / s	contraction of a vector
v + v	vector addition
v - v	vector subtraction
v * v	vector cross product
r * v	rotation of a vector
t * v	transformation of a vector
f * v	transformation of a vector - short hand for (stat ion →f)*v
v WRT f	a vector in stat ion coordinates pointing the same way as v points in f's coordinate system, v WRT f ≡ORIENT(f)*v ≡ (f*v) - POS(f)
UNIT(v)	vector of unit length pointing in the same direction as v
POS(f)	vector position of frame or trans
AXIS(r)	axis of rotation

Rotation operators

ROT(v,s)	constructs rotation of s degrees about v
ORIENT{ f)	orient at ion of a frame or trans
r * r	composition of two rot at ions (the one on the right is applied first)

Frame operators

FRAME(r,v)	constructs frame of orient at ion r at position v
CONSTRUCT(v,v,v)	makes a frame: first vector gives the position, second a point on the x-axis, third is a point in the xy-plane
f + v	translation of a frame
f - v	translation of a frame
t * f	transformation of a frame
f * f	transformation of a frame - shorthand for (station →f)*f

Transform operators

TRANS(r,v)	constructs trans which will cause a rotation of r followed by a translation of v
f → f	transformation which maps from the first frame to the second
t * t	composition of two transes (the one on the right is applied first)
INV(t)	take the inverse of t

The operators in AL generally follow "normal" precedence rules, i.e., functions are evaluated first, followed by exponentiat ions before multiplications or divisions, which in turn are performed before additions and subtractions. The order of operation can be changed by including parentheses at appropriate points. In an expression where several operators of the same precedence occur at the same level, the operations are performed from left to right.

TABLE OF PRECEDENCE

functions, ⟨⟩,||, NOT
WRT → ↑
*/ . MAX MIN DIV MOD
f -
= ≠ < > ≤ ≥
A
V ⊗
=

4.2.7 Predeclared constants

PI = 3.14159... (can also be written as π)
STATION is a frame which has standard station coordinates
BARM is the location of the blue arm
YARM is the location of the yellow arm
BHAND is the distance between the fingers of the blue arm
YHAND is the distance between the fingers of the yellow arm
BPARK is the rest position for the blue arm
= FRAME(ROT(yhat,180*degrees),VECTOR(43.53,56.86,9.96)*inches);
YPARK is the rest position for the yellow arm
= FRAME(ROT(yhat,180*degrees),VECTOR(40,14,9)*inches);
TRUE and FALSE have the obvious meanings (TRUE = 1, FALSE = 0)
XHAT is VECTOR{ 1 ,0,0)
YHAT is VECTOR(0,1,0)
ZHAT is VECTOR(0,0,1)
NILVECT is VECTOR{ 0,0,0)
NILROT is ROT(zhat, 0 * DEG)
NILTRANS is TRANS(nilrot,nilvect)
CRLF is a string constant that prints as a carriage return followed by a line feed

4.2.8 Some examples

```
DISTANCE VECTOR v 1,v2; {some declarations)
ANGLE SCALAR t hcta;
SCALAR ARRAY s 1 [ 1:5],s2[-3:3,1:2];
FRAME f 1 ,f2;
EVENT ready;

ROT(zhat,90*deg) * v l    {v1 rotated 90 degrees about
                                 the station's Z axis)
v1 , yhat              (the Y component of v1}
fl * xhat              {f1's X axis in station coordinates}
3 * s1[2]              {the second element of the
                                 array sl multiplied by 3)
```

4.3 Affixment: AFFIX *& UNFIX*

The relationships between the various features of an object, and between different objects, may be modelled by use of the *AFFIX* statement. The general form for the *AFFIX* statement is:

AFFIX fl TO f2 BY t AT <expr> <affix type>

The effect of the above is to establish a trans that expresses the relationship between *f1* and *f2*. If *<BY t>* is present the resulting trans will be associated with the variable *t* making the affixment relation modifiable by the user, otherwise an internal variable' will be created. The initial value of the trans is specified by the *<AT expr>* part of the statement. If none is given then the current values of *f1* and *f2* are used to create a trans taking *f2* to *f1* $(f2 \rightarrow f1)$. There are two flavors of affixment possible, and *<affix type>* specifies whether the affixment is to be done *RIGIDLY* or *NONRIGIDLY*. Rigid affixment is symmetric; when either frame is given a new value the other is updated to preserve the relationship between them, Non-rigid affixment is asymmetric; when *f2* is changed, the value of *f1* is updated, whereas when *f1* is modified, the trans describing the relationship between *f1* and *f2* is recomputed to express the new relationship between them. An example of non-rigid affixment would be a plate on a tray; the plate moves with the tray, but not vice versa. If *<affix type>* is not specified, rigid affixment will be assumed.

An affixment relation can be broken by use of the *UNFIX* statement:

UNFIX f 1 FROM f2

4.4 Motions

4.4.1 Compile-time and runt ime considerat ions

In the current AL system, trajectory calculation is done at compile-time. When the compiler encounters a motion. statement a trajectory is computed to accomplish it as fast as possible, subject to the constraints of maximum acceleration and torque imposed by the motors, using the compile-time planning values for all the relevant expressions that describe the requested motion. During actual execution these expressions may have different values, so the runtime system modifies the trajectory immediately prior to executing it. There are limits to how large a discrepancy can be corrected at runtime. If the planning value is seriously in error, then the at tempt to make last-minute correct ions rnight overstrain the arm, causing the motion to be aborted. One simple way of correcting this problem is to tell the compiler to take more time for the mot ion. Work is underway to implement runtime path calculation which will avoid this situation.

The compile-time trajectory calculator will issue error messages if an illegal motion is requested, such as trying to move to a position inaccessible to the arm, or requesting the motion to take less time than physically possible. It should also be noted that many of the parameters to the clauses modifying the motion must be constants.

4.4.2 The basic *MOVE* statement

The basic *MOVE* statement is of the form:

MOVE <controllable frame> TO <dest> <modifying clauses>

which will cause the specified arm to be moved so it has the same position and orientation as the destination frame expression *<dest>*. A grinch sign, "⊗", can be used in *<dest>* to represent the current position of *<control/able frame>* when the motion is executed, *<Controllable frame>* may be either an actual manipulator (*barm* or *yarm*) or a frame which has been affixed to one of the arms. In the latter case, the physical relationship between the frame and the arm, described by the affixment chain connecting them, will be used so the motion results in the frame being moved to *<dest>*. The motion may be modified in many different ways through the use of the various *<modifying clauses>* described below.

44.3 intermediate points: *VIA*, *DEPARTURE & APPROACH*

in the case where a motion must go through a series of intermediate points (to avoid obst acles, for inst ance), the intermediate frames may be specified by means of a *VIA* clause, such as:

```
VIA f1,f2,f3,f4,f5
```

where *f1,...,f5* are frame expressions. The motion will pass through the points in the order they are specified. it is also possible to specify the arm's velocity at a via point, and the duration of the motion from the last given point to the via point. This full *VIA* clause looks as follows:

```
VIA f WHERE VELOCITY = <v>, DURATION = <n>
```

where v is a velocity vector and *n* is a time scalar. One or both modifying clauses may be present, in either order. Note that unlike the first mentioned form, only one frame *f* may be given in this format, if the trajectory calculator believes that more than *n* seconds are required for this segment of the motion an error message will be generated. Both the velocity and duration values must be compile-time constants.

It is also possible to specify deproach points, which are points associated with departure of the arm from its current location, or its approach to the destination location, Unlike via points, deproach points are expressed with respect to the initial or destination coordinate systems. The clauses are as follows:

```
WITH DEPARTURE = <exp>
```

and

```
WITH APPROACH = <exp>
```

where *<exp>* may be as follows. Depending on whether the *APPROACH* or *DEPARTURE* clause is used, *<fr>* represents either the destination frame or the current posit ion.

type of *<exp>*:	deproach point in stat ion coordinates:
frame	<fr> * <exp>
vector	<fr> + <exp> WRT <fr>
scal ar	<fr> + (<exp> * zhat) WRT <fr>

it is also possible to indicate that no deproach point is to be used by specifying *<exp>* as *NILDEPROACH*, or to use the deproach point associated with some other frame using the function *DEPROACH(<frame id>)*.

Deproach points may be specified implicit iy. The statement:

DEPROACH(<frame id>) ← <exp>

will associate *<exp>* as the deproach point for the variable *<frame id>*. if then no approach point is explicitly given in the move statement, the deproach point associated with the destination frame will be used for the approach. if the destination frame does not have a deproach point associated with it, the compiler will search along the frames affixed to it until a deproach point is found. If none are discovered then the deproach of the station *(3 * zhat * inches)* will be used. If the destination is a frame expression then *NILDEPROACH* will be the default approach used. if no departure point is specified, then the approach point for the last move will be used.

The AL predeciared macro *DIRECTLY* expands into the two clauses:

```
WITH DEPARTURE = NILDEPROACH
WITH APPROACH = NILDEPROACH
```

4.4.4 Force & Compliance

It is possible to have the arm apply or sense specified forces and moments. (Sensing forces is discussed in section 4.4.5 below.) To avoid incompatible requests the force components must always be orthogonal. To insure this, a force frame must be specified, and the directions of the applied forces and moments must be aligned with one of the cardinal axes of this current force coordinate system. Also specified is whether the orientation of the axes changes as the hand moves, i.e. is the force frame defined relative to the hand or the table (world) coordinate system. The clauses to do all this are as follows:

```
            WITH FORCE = <sval> ALONG <axis-vector> OF <frame>
                                               IN <coord sys>
            WITH TORQUE = <sval> ABOUT <axis-vector> OF <frame>
                                               IN <coord sys>
. or
            WITH FORCE-FRAME = <frame> IN <coord sys>
            WITH FORCE = <sval> ALONG <axis-vector>
            WITH TORQUE = <sval> ABOUT <axis-vector>
or
            WITH FORCE-FRAME = <frame> IN <coord sys>
            WITH FORCE(<axis-vect or>) = <sval>
            WITH TORQUE(<axis-vector>) = <sval>

where:      <axi s-vect or> = xhat, yhat or zhat.
            <coord sys> = HAND or WORLD (default = WORLD)
```

<sval> = the magnitude of the force
<frame> = the orientation of the axes of the force frame

in the first form the specified force frame in ail of the clauses must be the same. if *IN <coord sys>* is not specified, WORLD is assumed, while if *OF <frame>* is omit t ed, *STATION* is assumed. Note that only one force frame may be specified per move. Applying a force of magnitude zero means that the arm will be compliant, i.e. move away from any external force in that direction,

A short form is also available for those motions which only need to apply or sense one force, but not both. It looks like either;

```
WITH FORCE = <sval> ALONG <vect> OF <frame> IN <coordsys>
```

or

```
WITH FORCE(<vect>) = <sval>
```

This generalizes in the obvious way for *TORQUE* and for force sensing. if no *<frame>* and *<coord sys>* are specified then a force frame in world coordinates is automatically created with it's x-axis aligned along *<vect>*. Otherwise the specified coordinate system is used and a force frame is created with it's x-axis along *<vect>* WRT *<frame>*.

4.4.5 Condition monitors

4.4.5.1 Types: force, duration, event & boolean

During the course of an arm motion it may be desired to monitor some condition, or set of conditions, and to execute an act ion if the condition has occurred, The condition monitor clause is used for this purpose. It has the following general form:

```
ON <condition> DO <action>
```

Currently the conditions that can be monitored include force sensing, duration, events, and various boolean expressions of variables. *<Action>* may be any valid AL statement or block. The only restriction is that if a motion statement is the only statement in *<action>* then it must be surrounded by *BEGIN* and *END* to prevent ambiguity.

The monitoring will begin with the start of the motion and continue until the mot ion terminates. if the monitor triggers, then after it finishes its action, it will become dormant and cease checking its condition. It is possible to modify this by use of the *ENABLE* and *DISABLE* statements described below (section 4.4.5.2).

When sensing forces and moments the following clauses are used:

```
                ON FORCE <rel><sval> ALONG <axis-vector> OF <frame>
                                                IN <co-ord sys> DO <action>
                ON TORQUE <rel><sval> ABOUT <axis-vector> OF <frame>
                                                IN <co-ord sys> DO <action>
or
                WITH FORCE-FRAME = <frame> IN <co-ord sys>
                ON FORCE <rel><sval> ALONG <axis-vector> DO <action>
                ON TORQUE <rel><sval> ABOUT <axis-vector> DO <action>
or
                WITH FORCE-FRAME = <frame> IN <co-ord sys>
                ON FORCE(<axis-vect or>) <rel><sval> DO <act ion>
                ON TORQUE(<axis-vector>) <rel><sval> DO <action>
```

where: *<axis-vector>*, *<co-ord sys>*, *<sval>* and *<frame>* are the same as in section 4.4.4 above and *<rel>* is either ≥ or <, the condition monitor triggering when the force or moment exceeds or goes below the specified magnitude respectively. As in applying forces there is a short form when only one force is being sensed or applied:

```
                ON FORCE <rel><sval> ALONG <vect> OF <frame>
                                                IN <co-ord sys> DO <action>
or
                ON FORCE(<vect>)<rel><sval> DO <action>
```

The condition monitor:

```
                        ON DURATION ≥ n * seconds DO <action>
```

will trigger its action *n* seconds after being enabled at the start of the motion.

```
                        ON <event> DO <action>
```

means do the action if *<event>* is signailed (by another condition monitor or some other parallel process).

```
                        ON <boolean expression> DO <action>
```

has the effect of evaluating the boolean expression, made up of algebraic variables, and if it is true (non-zero) performing the desired action. If the expression is false the condition monitor goes to sleep for a short while (currently 100 milliseconds) before evaluating and checking the expression again.

4.4.5.2 *ENABLE* and *DISABLE* -labelled condition monitors

A condition monitor has two states: enabled and disabled. in the enabled state it will trigger its conclusion if the condition it is checking for occurs, in the disabled state the condition monitor is inactive. As mentioned above a condition monitor is enabled when the motion is started,--and disabled upon the conclusion of the rnot ion. Once a condition monitor triggers it will become disabled, unless it is explicitly reenabied. This reenabling is done by means of an *ENABLE* statement placed in the conclusion of the condition monitor.

With the *ENABLE* and *DISABLE* statements it is possible to change the state of an arbitrary condition monitor that has been named by putting a label immediately before the reserved word *ON*. The syntax of these statements is:

```
ENABLE <condition monitor>
```

and

```
DISABLE <condition monitor>
```

Prefacing a condition monitor with the reserved word *DEFER* will cause it to be initially disabled. it can then be explicitly enabled later. Here is an example where a condition monitor is initially disabled, and then after three seconds is enabled:

```
MOVE barm TO dest
   test: DEFER ON FORCE(zhat)≥10* oz DO STOP
         ON DURATION ≥3*sec DO ENABLE test
```

4.4.6 Other clauses: *DURATION,SPEED_FACTOR,NULLING& WOBBLE*

Here are some other clauses that can be used to modify motions. Note that the parameters of these clauses are compile-time constants.

```
WITH DURATION = <sval>
```

causes the resulting motion to take the amount of time specified by *<sval>*, which should be of dimension *TIME*. if the trajectory calculator thinks that more time is needed AL will issue a warning message.

```
WITH SPEED-FACTOR = <sval>
```

slows down the motion. The minimum time for the mot ion computed by AL will be mult ipiied by *<sval>*, which should be ≥1, and this product will be used as the time for the motion.

The default speed factor for motions is 1, so the motion takes as little time as possible. This can be changed by assigning the desired default multiplier to the predeclared variable *SPEED_FACTOR* with a regular assignment statement:

SPEED-FACTOR ← <new default speed factor>

There are also two predefined macros: *CAUTIOUS* and *SLOW,* which set the default speed factor to 2 or 3 respectively.

WITH NULLING

informs the runtime system to null out errors at the end of this motion. There is also a *WITH NO_NULLING* clause which is the current default. There are two macros PRECISELY and APPROXIMATELY which achieve the same results.

WITH WOBBLE = <sval>

adds a small sinusoidal motion to the outer three joints causing them to shake a bit. It is useful-for breaking small friction forces and for seating parts. *<Sval>* is a small compile-time constant of dimension *ANGLE* that is usually about 2 or 3 degrees.

4.4.7 Controlling the fingers: *OPEN, CLOSE & CENTER*

The fingers can be controlled in several ways.

OPEN <hand> TO <sval>

and

CLOSE <hand> TO <sval>

causes the fingers to open or close so that they are a distance *<sval>* apart. *<Sval>* is any scalar expression of dimension *DISTANCE.* Currently there is no difference between the *OPEN* and the *CLOSE* statement. Eventually *CLOSE* will stop the motion of the fingers if both touch sensors are triggered.

CENTER <arm>

closes the fingers of the specified arm until both touch sensors indicate contact has been made. Furthermore if one finger makes contact before the other, *CENTER* causes the arm itself to move so that the object being grasped is not pushed by the finger. *OPEN* and *CLOSE* only move the fingers, and if the object being grasped is not centrally located between the fingers, the object will be moved or, if it is fixed in place, excessive force might be exerted by the fingers, thereby aborting the mot ion.

4.4.8 *STOP* & *ABORT*

There are two ways of terminating motions before they finish:

STOP <device>

and

ABORT(<print list>)

The *STOP* statement causes the indicated device to stop. <Device> may be a physical manipulator or a frame affixed to an arm. If *<device>* is not specified, and the stop statement appears in the scope of a move statement, then the arm used for the motion will be the one stopped. The *ABORT* statement is used for more drastic occasions. It will stop the motion of all devices, print out the elements of the *<print list>* (see the description of the *PRINT* statement, section 4.5.7, below), and transfer control to 11DDT. The user may continue the program execution by typing <alt>P to 11 DDT. Usually these statements appear in the body of condition monitors, though they may be appear at any point in the program.

4.4.9 Other devices - the *OPERATE* statement

The *OPERATE* statement is provided to control other devices interfaced to the AL system. Its syntax is similar to that of the *MOVE* statement:

OPERATE <device> <modifying clauses>

where *<device>* is the device being controlled, and the *<modifying clauses>* describe what action the device shall perform. For example;

OPERATE vise WITH OPENING = 4 * inches

Currently *no* devices other than the arms are available. A screwdriver and vise will be available soon,

4.5 Non-mot ion statements

4.5.1 Assignment statements

The assignment statement;

<variable> ← <expression>

causes the value represented by *<expression>* to be assigned to the variable appearing to the left of the assignment symbol. The data type and physical dimension of the expression on the right hand side of the assignment symbol must be the same as the data type and dimension of the variable on the left hand side.

There is also another form, the plan-time assignment:

<variable> ←← <expression>

where *<variable>* and *<expression>* are the same as above. The plan-time assignment statement provides a way of passing to the world modeller the values of certain variables whose values it would not otherwise know until runtime (e.g. *barm, yarm, bhand* and *yhand*). No executable code is generated for plan-time assignments. They only have an effect during program compilation.

An example of an instance where the plan-time assignment would be necessary is after a move statement that will terminate early (e.g. stopping on touch). A better trajectory can be computed by using a plan-time assignrnent to pass the trajectory calculator the expected position of the arm at the end of the motion. During runtime the actual value of the manipulator will be determined by the physical world, and this value will be used to modify the computed trajectory.

4.5.2 Traditional control structures: IF, *FOR, WHILE, UNTIL, CASE*

AL has many of the traditional ALGOL cont rol structures.

The IF statement has the form;

IF <boolean expression> THEN <statement> ELSE <statement>

The *ELSE* part is optional. If *<boolean expression>* is true (non-zero> the statement following the *THEN* is executed. Otherwise the statement following the *ELSE*, if present, will be executed.

The *FOR* loop has the form:

FOR <svar>←<sexpr> STEP <sexpr> UNTIL <sexpr> DO <statement>

where *<svar>* is a scalar variable and the *<sexpr>*'s are scalar expressions of the same dimension. The initial value of the variable is the value of the first expression; every time the statement is executed, its value is incremented by the value of the second expression, and the process repeats until the value exceeds that of the third expression. if the step size is negative, the right things happen. The test is made before the first iteration, so it is possible that the loop will not be executed at all.

The *WHILE* loop is as follows:

WHILE <boolean expression> DO <statement>

The boolean expression is checked and if it is true the statement is executed. This process is repeated until the condition becomes false.

The *UNTIL* statement is as follows:

DO <statement> UNTIL <boolean expression>

where the statement is repeatedly executed until the condition becomes true. This is similar to the *WHILE* statement described above, with the exception that the *WHILE* loops while the condition is true, whereas the *UNTIL* loops until the condition is true.

There are two forms that the *CASE* statement may take. The regular *CASE* statement has the form:

CASE index OF BEGIN SO; S1; S2; . . . Sn END;

The index is evaluated and depending on the integer part of its value one of the statements will be executed. if the index is zero then SO is chosen, if the index is one then S1 is chosen, and so on up till n. if the index is negative, or greater than the nurnber of statements, an error is reported. Any of the statements may be null, e.g. "Sl;; S3", in which case if the index were two no statement would be executed.

There is also a numbered version of the *CASE* statement:

CASE index OF BEGIN [CO] S; [Cl] [C2] S; ... [Cn] S; ELSE S END

where each stat ernent has one or more non-negative scalar constants labeiling it. - The index expression is again evaluated and if it is the same as one of the Ci's then the statement with that label is executed. If no constant matches the index then nothing is done, unless an *ELSE* is present in which case the statement it labels is executed. If the index is negative or greater than the largest Ci an error occurs, unless there is an *ELSE* present. Note that the *ELSE* statement may appear anywhere in the list of statements, not necessarily at the end.

4.5.3 Procedures

Procedures are defined as follows;

<type> PROCEDURE <name> (parameters);
<st at ement>;

where the statement is executed each time the procedure is called. Only those procedures that return a result need their type specified. The data types of the parameters may be modified by the reserved words: *VALUE & REFERENCE*. Reference is the default. Due to the world modelling performed by the AL compiler it is necessary when defining a procedure to specify the dimensions of any arrays that are to be used as a parameter. For example:

```
PROCEDURE foo(FRAME ARRAY pnt s[1:4,1:3]);
```

It is also necessary to make plan time assignments to any formal parameters or variables that are defined in the same block as the procedure so that when the procedure is simulated the values will be available (c.f. section 4.6).

Procedures can return a result by means of the *RETURN* statement which has the form:

```
RETURN (value)
```

which returns value as the result of the procedure. The *RETURN* statement may not appear inside condition monitors or *COBEGIN-COEND* blocks.

Procedure calls take the normal form of the procedure name followed by the list of arguments: name(arglist). They may appear anywhere an expression might, or alone by themselves as a procedure statement. If a typed procedure appears in a procedure statement then the result it returns will be discarded.

4.5.4 Parallel control: *COBEGIN-COEND, SIGNAL & WAIT*

In addition to the normal sequential execution of statements within a *BEGIN-END* pair, AL allows blocks of code to be executed in parallel by placing them in a *COBEGIN-COEND* block. Upon entering the *COBEGIN* block control is divided among the various processes to be executed simultaneously. Upon the termination of all of these processes control will be passed to the part of the prograrn following the *COEND*. it is the user's responsibility to ensure that the code being executed in parallel is sufficiently independent (e.g. two processes don't try to use the same arm at the same time), and that no deadlock situations occur.

it should be noted that the purpose of the *COBEGIN* construct is to allow simultaneous independent manipulator cont rol. It is not particularly useful to execute purely comput ational code in parallel, though doing computation while an arm is moving can save time. The scheduling algorithm used is to start up one process and execute it until it is blocked, and at that point another process will be run. A process can be blocked by waiting for an event, by pausing, doing I/O, or by initiating a motion.

Parallel processes may be synchronized by means of explicit events and *SIGNAL* and *WAIT* statements. With each event is associated a count of how many times it has been signalled. initially, the count is *zero,* that is, no signals have appeared, and no processes are waiting. The statement:

SIGNAL el

increments the count associated with event *el*, and if the resulting count is *zero* or negative, one of those processes waiting for *el* is released from its wait and readied for execution, The statement;

WAIT el

decrements the count associated with event *el*, and if the resulting count is negative, the process issuing the WAIT is blocked from continuing until another process signals *el*. If the count is zero or positive, there is no waiting.

4.5.5 Statement condition monitors

Condition monitors, besides modifying motions, may also appear as statements, The description in section 4.4.5 also applies to statement condition monitors+ When its defining statement is executed the statement condition monitor will become enabled. It will become disabled when it triggers, is explicitly disabled (it must be labelied for th'is to occur), or its local block is exited. The reserved word *DEFER* still causes a condition monitor to be defined in an initially disabled state,

Scope rules come into play regarding when condition monitors may be enabled or disabled. An enable or disable statement may only refer to a condition monitor that is defined in the same block as itself or in a block containing it.

4.5.6 *PA USE* stat ement

The statement:

PAUSE <sval>

will result in the program going to sleep for the time specified by *<sval>*, which should be of dimension *TIME.*

4.5.7 I/O

At runtime strings and variable values may be typed out using the *PRINT* statement:

PRINT(<arg1>,<arg2>,...,<argn>)

where the *<arg>*'s are either algebraic expressions or variables, or string constants. Strings are delimited by double quotes. *CRLF* is a predefined string which prints as a carriage return followed by a line feed.

The statement:

PROMPT(<print list>)

is syntactically like the *PRINT* and *ABORT* statements. Upon encountering a PROMPT statement the AL runtime system prints out all the items in the print list and then prints the message:

"Type P to proceed"

and waits for a P to be typed. Unlike the ABORT statement control does not pass to DDT and hence any parallel processes (e.g. ALAID or COBEGIN) will continue to be executed. As an example;

```
PROMPT("Move barm to work station origin"); org ← barm;
```

There are two arithmetic operators to read in a value from the VT05 console. *INSCALAR* reads in a scalar, prompting the user with; "SCALAR, please: ". *QUERY* reads in a boolean. It is like *PROMPT* in that it can have a print list. After typing the print list the user is asked to "Type Y or N: ". For example:

```
PRINT("How tall is casting?"); height ← INSCALAR;
WHILE QUERY("More to do?") DO . . .
```

ALAID, a debugger for AL, may be used to do I/O between AL and another program (usually one doing a vision task on the PDP-10). (c.f. section 7.6.3.1).

4.5.8 Macros

AL possesses a general purpose text macro facility. The syntax for a macro definition is:

DEFINE <macro id> <parameters> = ⊂<macro body>⊃

where *<macro id>* is the name of the macro, *<macro body>* is the text to be substituted whenever *<macro id>* is encountered in the program, *<parameters>* if present is a list of arguments for the macro, separated by commas and enclosed by parenthesis. Only undeclared identifiers may be used as macro parameters. When the macro is expanded the actual arguments will be substituted into the macro body wherever the parameters appear, If this value is anything other than a simple token it must be surrounded by the delimiters ⊂⊃. The *<macro body>* is also delimited by ⊂⊃.

Here are two examples of the use of macros:

```
DEFINE feet =⊂12* inches>;
DEFINE grasp( frob) = ⊂MOVE barm TO frob;
                       CENTER barm;
                       AFFIX frob TO barm RIGIDLY⊃;

size ← 10.4 * feet; {Expands to 10.4 * 12 * inches)
grasp( handle);     {Expands to;
                     MOVE barm TO handle;
                     CENTER barm;
                     AFFIX handle TO barm RIGIDLY;)
```

4.5.9 *REQUIRE* statement

REQUIRE statements allow the user or his program to communicate with the AL compiler. No code is generated as a result of a *REQUIRE* statement, and the effect of the *REQUIRE* statement is global and persists after exiting the block in which it was invoked. Another *REQUIRE* statement or some other termination condition is necessary to undo or stop the effect.

REQUIRE SOURCE-FILE "<file_name>"

The file named will be the source of future input until an end of file is encountered, at which time the code following the require will be read. The source file will be assumed to be a disk file, unless specified as a teletype file by "TTY:" in front of its name.

A teletype file does not need a name, but if it has one, the teletype input will be saved on a disk file with the given name and default extension TTY, Parsing action on teletype inputs will begin each time a carriage return is hit. The file is closed by typing a <control><meta><linefeed>. The current operating system allows only one teletype file to be open at a time.

The file name can be one of:

```
"NAME"
"NAME.EXT"
"NAME[P,PN]"
"NAME.EXT[P,PN]"
```

where P and PN represent the project and programmer names respectively.

REQUIRE MESSAGE "<message>"

Anything appearing within the double quotes will be printed out at the user's t erminai.

REQUIRE ERROR-MODES "<mode flags>"

While the AL parser may ask for user responses to errors during program compilation, it is possible to predefine the standard treatment of errors by setting certain flags with the *REQUIRE ERROR_MODES* statement, The flags are set by including the relevant letter within the quotes, and reset by including a minus sign in front of the code letter. The following flags are available;

L	-	errors, if any, will be logged in a file with extension LOG
A	-	compilation will continue automatically after each error message is printed.
M	-	the system will prompt the user only for modifiable errors
F	-	strict dimension checking will not be carried out across assignment statements, condition monitors, etc. Undimensioned variables will be coerced according to the cont ext in which they appear. Error messages will be generated only for inconsistent usage.

REQUIRE COMPILER-SWITCHES "<compile switches>"

Ail the switches that are used in the command line (see Chap 5) can be . specified here. This is an alternative to specifying the switches in the command line. Only letters (without the slash) should be within quotes.

4.5.10 Debugging aids: *NOTE & DUMP*

To facilitate tracking down errors that are reported by the AL compiler the following two statements may be used. Their action is only at compile time, and no code is generated for them,

The *NOTE* statements result in some message being output during compile time. *NOTE 1* will output the message during the world modeiiing phase, while

NOTE 2 will output the message during the code emission and trajectory calculation phase. *NOTE* will output the message during both of these phases.

The syntax is as follows:

```
NOTE("message")
NOTE 1 ("message")
NOTE2("message")
```

The *DUMP* command will print the planning values of the desired variables at the point in the program that it is encountered. It looks like this:

```
DUMP <id list>
```

where *<id list>* is a list of identifiers separated by commas,

4.6 World modelling

As mentioned earlier (section 4.4.1) the current AL system does trajectory calculation at compile-time. To accomplish this the compiler must have a model of the world containing the position of the arm, the expected values of the variables specifying the motion, and knowledge of the affixment structure. These values are different at different points in the program, so the compiler must perform a simulation of the program in order to obtain the information required for the trajectory calculator. An AL program goes through several stages during its compilation: first it is parsed and an internal representation built, then the program is simulated (in the world modelling phase), and finally trajectories are calculated and code is emit ted. Certain problems arise in doing the world modelling due to the unavailablity of sensory data at compile-time. Also various compromises have been made in the simulation of loops and parallel processes.

During the world modelling each statement has an input world and an output world associated with it. The input world gives the current position of the arms, the value of each variable, and the affixment structure immediately prior to the statement's execution, while the output world reflects the effect of the statement. Most statements (e.g. assignment, affixment) are quite easy to sirnulate. Motion statements that do not have any associated condition monitors are also easily dealt with. (The problem of collision avoidance is not currently handled,) However, if a condition monitor is present the world modeller will be unable to determine whether or not it is ever triggered, and hence be unable to judge its effect on the motion. For example, if the arm is holding a box and we wish to set it down on a table, the code might look like this:

```
MOVE box TO table - 4 * zhat * inches
      ON FORCE(zhat) ≥ 8 * oz DO STOP;
```

We expect the motion to be stopped short when the box encounters the table, but the world modeller, even if it knows this, cannot determine the position of the box when the condition monitor triggers, Because of this the current system ignores the effect of any condition monitors associated with a motion, and assumes that the motion terminates normally. Fortunately for purposes of planning trajectories, the inaccuracies introduced in this way are generally unimportant. For those cases where the effect of the condition monitor is known ahead of time to be critical, the plan-time assignment statement should be used to inform the world modeiier.

The next type of statements that give the world modeller trouble are conditionals. A very liberal approach would assert that any fact that is true in either one of the output worlds of the conditional branches, except for those that conflict, should be considered true. AL currently takes a more conservative approach and treats as true only those facts that are true no matter which way control goes. The effect of this is that after a conditional statement like;

```
IF j>5 THEN j←1 ELSE j←j+1
```

the variable *j* will have two different values depending on which half of the conditional was executed, and as a result the world modeller will not have a planning value for the variable *j*. If one is needed for later code then a plan-time assignment statement giving *j* an expected value should follow the conditional. The CASE statement is handled in a similar fashion.

Loops also present difficulties. The world modeller will unroll the loop one iteration. Note that only one trajectory will be computed for each MOVE in the loop. This can present problems if the destination of one of the moves takes on drastically different values, In such a case the user rnay need to include code to choose from several move statements, based on the possible destination values, or to allocate more time to the move so the arm will have enough time to complete the longest of the moves.

As might be expected the COBEGIN construct is also hard for the world model ler. The way AL handles parallelism is to combine all of the changes introduced in each branch, excluding any obvious incompatibilit es such as the same variable being assigned different values by different branches. All of these facts taken together then form the output world for the COBECIN block.

Procedures present several problems. The body of a procedure is modelled once upon entry of the block in which it is defined. It is necessary to make plan time assignments to any formal parameters or variables that are defined in the same block as the procedure so that when the procedure is simulated the values will be available. Also note that the same problem that occurs when using MOVE statements in a loop also applies to procedures, i.e. only one trajectory is calculated at compile time, and if the actual motion at runtime is drastically different the motion will fail. Procedure calls are essentially ignored; they have no effect in the world model. Typed procedures return a result of zero (nilvect, nilrot, etc.) as far as the world modeller is concerned.

The world modeller initializes the input world of the first statement in the program so that the arms are in their park position, the fingers are two inches apart, and the speed-factor used by motions is one. If the arms in the output world of the last statement are not again in their park positions the world modeiier will issue a warning message.

5. USING AL

This chapter describes the steps involved in compiling and executing an AL program if there are no errors. Should there be any error messages the reader should refer to Chapter 7 to find out what to do about them. In the following description where commands are typed by both the user and by the system, the system response will be shown in italics,

5.1 Compilation of user programs

To compile and prepare the binary load module for the PDP-11 do the following:

1. Create a file called "FOO.AL" with your program in it, where "FOO" may be any name you wish.

2. Get your job to monitor level and type "COMPILE FOO".

2a. The system program SNAIL which handles requests like COMPILE will give the message

Swapping to SYS: AL. DMP

and then start AL at the parser. The parser will then say

AL: FOO

When the parser hits a page boundary in your file, it will type "*1*" or whatever the number of the page that it is starting to read.

2b. When the parsing is complete, the parser swaps to the AL compiler, which types "*ALC*".

2c. When the compiler completes its world modelling, trajectory calculation, and code emission, it swaps to the cross-assembler PALX for the PDP-11. "*PALX n*", where *n* is the version number of the PALX compiler, is typed out at the user terminal.

2d. The PALX compiler swaps to ALSOAP, which cleans up the user area by deleting the intermediate files with extensions .ALP,.ALV,.ALT, and SEX that are created during the compilation of the AL program.

2e. The job gets back to monitor level.

If you misspell the name of your file then SNAIL will complain

File not found: FOO

where "FOO" is your misspelling.

At any time during 3 through 6 above, you could get an error message from the parser, compiler or PALX. See Chapter 7 about these.

5.1.1 Compilation with switches

Compilation may be done with switches if desired by including the desired switches within parentheses as "COMPILE FOO.AL(KS)". Effects of the different switches are shown below:

K	Keep the intermediate files (.ALP,.ALV,.ALT)
S	Inhibit deletion of the SEX file
L	Generat-e a PALX assembly listing

5.2 Loading and executing the AL program

When your program "FOO.AL" has got through to ALSOAP without grief, you are ready to execute the program on the PDP-11.

1. Locate the brake control box(es) for the arm(s), and the position{ s) of the panic butt on(s). Keep your finger poised over the panic button at all times while the AL program is being executed (procedure for starting it is in step 3), and be prepared to press it immediately if it should appear that something unpredictable or disastrous is about to happen. Pulling the yellow cord that runs around the table will turn off power to the arm, and can also be used in the event of an emergency. Take care not to lean on the cord accidentally.

2. Type "DO AL[AL,HE]" followed by carriage return. This initiates a series of instructions which are described later (5.3). When you see

f=

type "FOO", the name of your program, followed by carriage return. You will then see a number of lines printed out, The last line will be

DDT STARTED AT 130000

3. Now go to the VT05 which is a white colored terminal with a dark screen in the area of the hand-eye table, and on it, you should see an asterisk "*" and a flashing cursor. Make sure you have the panic button under your thumb and then type

```
* START <alt><alt> G
```

to begin execution of your program. Note that just <alt>G will also work. AL will print out at the VT05:

```
AL RUNTIME SYSTEM
```

The VT05 will beep just before the start of each motion by the arm. Messages or values will be printed where appropriate. When program execution is complete, the following message will appear;

```
ALL DONE NOW. SEE YOU AROUND!
NO ACTIVE PROCESSES LEFT. YOU'RE IN DDT.
*
```

Type "<alt>G" at the VT05 to re-execute the program from the beginning.

5.3 Complete runt ime execution sequence

The following is the complete sequence of operations required to load and execute an AL program once a binary file has been prepared. It is given in case some error occurs when the user types a DO AL[AL,HE].

1. Type "A ELF" to have the ELF (PDP-11 interface) assigned to your job, so that some other job will not try to use it while you are running your program.

2. Type "R11TTY" to execute the program that loads your program into the POP-1 1. 11TTY will respond with

```
CORE SIZE = 2SK
VERSION USING <device>
TYPE ? FOR HELP
*
```

where device is either VT05 or TERMINAL. The asterisk is 11TTY's way of prompting for user input,

The way to change (toggle) between the two devices is to type "V" immediately after the asterisk, and 11TTY will fill in the rest of the line and ask for the next prompt as follows:

```
*VERSION USING <other device>
*
```

An alternate way to get the device of your choice is to use an extended command by typing “A” followed by "VT05" or “TERM” to select the desired device.

```
*AN EXTENDED COMMAND VT05
*
```

It is desirable to use the device VT05 so that once execution starts, you can be totally independent of the PDP-10 (you may need to do so if you are running ALAID).

3. Type "Z" to zero out the core, followed by the memory size, currently 500000, then a carriage return to confirm the instruction, 11TTY will respond as follows:

```
*ZERO CORE [CONFIRM] 500000<cr>
*
```

4. The AL interpreter and the runtime system is then loaded by typing "G" for getting the core image binary file, followed by the name of the file AL[AL,HE] and a carriage ret urn.

```
*GET SAV FILE - AL[AL,HE]<cr>
*
```

5. The user’s AL prograrn binary file is then loaded by means of typing "O" for overlay, followed by the name of the file and a carriage return.

```
*OVERLAY BIN FILE - FOO<cr>
*
```

6. The next step is to get the program started by typing "S" then "D" followed by a carriage return.

```
*START AT (1000) (D FOR DDT) - D<cr>
 DDT STARTED AT 130000
*
```

7. Now go to the VT05 and after making sure you are ready to push the panic button type

```
*START <alt><alt>G
```

AL will print out at the VT05:

```
AL RUNTIME SYSTEM
```

Any other input or output will be typed at the VT05.

When program execution is done, the following message will be printed out at the VT05.

```
ALL DONE NOW.   SEE YOU AROUND!
NO ACTIVE PROCESSES LEFT. YOU'RE IN DDT.

*
```

Typing "<alt>G" on the VT05 will re-execute the program from the beginning.

6. POINTY

6.1 Description of POINTY

6.1.1 Introduction

The concept of *FRAMES* as a data structure in AL and their affixment to form an object model should be clear to the reader by now, The generation of such affixments of frames is a non-trivial task, especially if the frames to be affixed to each other have different orientations. If the object is physically available, the user would need to measure distances, angles, and positions, and by doing some rotation of frames would be able to determine the relationships bet ween the frames. Such a procedure is tedious and error-prone in all but the simplest cases.

Given the object, a means of generating the affixment structure is needed. The ideal case would be to present the physical object or its design drawing to the computer by utilizing vision, etc., and let the system build the affixment structure. However, the features of interest on the object are dependent on the nature of the assembly procedure, and may not bear any relationship to the shape of the parts. One way of generating an affixment structure is to use human assist ance. The human operator will point out the features of interest on the object, and the system will take care of the book-keeping involved in keeping straight the relationships between the various features.

The interactive construction of world model descriptions for AL programs has been achieved using POINTY, a system developed and implemented at SAIL. It makes use of the ability to read arm positions to define points of interest on the object.

By moving the manipulator around manually and reading the location, the user is able to record various positions on the object. He then tells the system how the various locations are related to each other so that an object model can be generated such that all the required features on the object are known once the position and orientation of one point is known,

POINTY provides the ability to do limited motion statements, This allows the user to try out various move statements before putting them into an AL program, permits the arm to be reoriented, and allows differential moves with the same orient at ion.

6.1.2 Pointing with a manipulator

6.1.2.1 Implicit specificat ion of frames

The Scheinman arm has 6 degrees of freedom, which allows it to be positioned at an arbitrary position and in an arbitrary orientation, Frames also have 6 degrees of freedom, corresponding to 3 components of translation and 3 angles of rotation. it follows that if a single pointing of the manipulator is to imply a unique frame explicitly, there are no spare degrees of freedom. The absence of spare degrees of freedom makes it quite difficult to position the manipulator accurately, since ail motions fine or gross require the movement of the same members, and also limits obstacle avoidance.

it is not difficult to guide the arm manually to a good grasping position to pick a part out of a fixture or pallet. it can be quite difficult to guide it manually to a good orientation such that when the manipulator attempts to remove the part, there is no binding. The need for orientation accuracy becomes more crucial when it is being used to define a world model, since any angular error may be multiplied by some long moment arm in the AL program.

To avoid this difficulty, it is sometimes convenient to use multiple pointings to define each frame implicitly. The first pointing may define the origin of the frame, the second may define one axis of the frame, and the third may define one plane of the frame. in this manner, each pointing determines position only, and there is no need to have orientation precision.

A simplification is possible when the orientation is parallel to that of some other known frame, e.g. station or some other predefined frame, in which case, the orient at ion frame can be specified from the known frame, and the iocat ion determined by means of a single pointing.

6.1.2.2 Pointer

The manipulator extremity must be provided with some sort of sharp pointer so that it can be used as a precise measuring tool. The pointer must have a shape suitable for reaching into awkward places such as the inside of a screw hole, the interior of a box, and so forth. in order to make the pointer shape compatible with ail kinds of unforeseen obstructions, it is desirable to design a pointer which may be bent by the user into an arbitrary shape. Such a special device will be referred to as a bendy pointer.

Whenever the user wishes, he may deform the bendy pointer into any new configuration which appears to be convenient for the next operation. Having deformed the pointer, the user must calibrate its new end position by using the pointer to point to a standard fiducial mark at a known location in the iaborat ory.

From the frame of the fiducial and the frame of the gripper, the system can infer t hc translation which takes the gripper frame into the bendy pointer. An alternative to the bendy pointer would be a tool set consisting of an assortment of rigid pointers of commonly useful shapes which could be quickly attached or detached. Whatever type of pointer is used, it must be reasonably rigid under gravity to prevent it deforming accidentally while being positioned.

6.1.3 System hierarchy

The POINTY system resides on two computers during execution - the PDP-l 0 where the arithmetic and computation is performed, and the PDP-11 which is responsible for reading and moving the manipulators.

The PDP- 10 part contains several modules: the affixment editor, arithmetic routines, manipuiat or interface, file input/output facilities, display routines, command line scanner (parser), and the user interface.

A subset of AL statements and expressions are accepted by POINTY. The command line scanner (parser) prompts the user for input of a new statement by an asterisk "*". if it is waiting for the continuation of a staternent or expression, it prompts with "****>>>". Parsing of the current input line begins when the user hits a carriage return. The first token of the input line is compared with entries in the symbol table. if there is a match, a fixed sequence of parsing will be followed, depending on the token. if no match is found, the parser checks to see if it is a variable that is on the left hand side of an assignment statement by checking to see if the next symbol is a back arrow "←".

The user interface communicates with the user by giving out error messages when the parser does not recognize something, or if the user wants to edit values of variables (e.g. orientation of frames) etc, without using an assignment statement.

Display routines update the screen of the user's terminal to reflect the current state of the affixment editor, arithmetic section, and the manipulator interface whenever the values change, or shut off the display altogether if necessary.

The affixment editor contains facilities for creating frames and modifying the relationships between them.

Arithmetic routines are called by the parser when it recognizes an expression or an assignment. it contains a fuii set of operations for *SCALARS, VECTORS, ROTS, FRAMES,* and *TRANSES.*

The file input/output facility contains routines for saving and restoring variables and values in and from a text file of AL declarations. These AL declarations and assignments may be used directly by an AL program, or they may be read in by POINTY if the user needs them for re-initialization to a known world state.

The manipuiat or interface provides the communication between POINTY and the manipuiat or, It allows communication with the runtime system residing on the PDP-11, and has two main functions: to transfer arm joints and wrist readings from the PDP- 11 to the PDP- 10, and to transfer commands for movement from the PDP-10 to the POP-1 1.

The PDP-11 part of the POINTY system essentially consists of the AL arm code, and a program which reads the arm position, continually displays it on the VT05, moves the arms when requested to do so by the PDP- 10 part, and prints out the result of each attempt to move the arm.

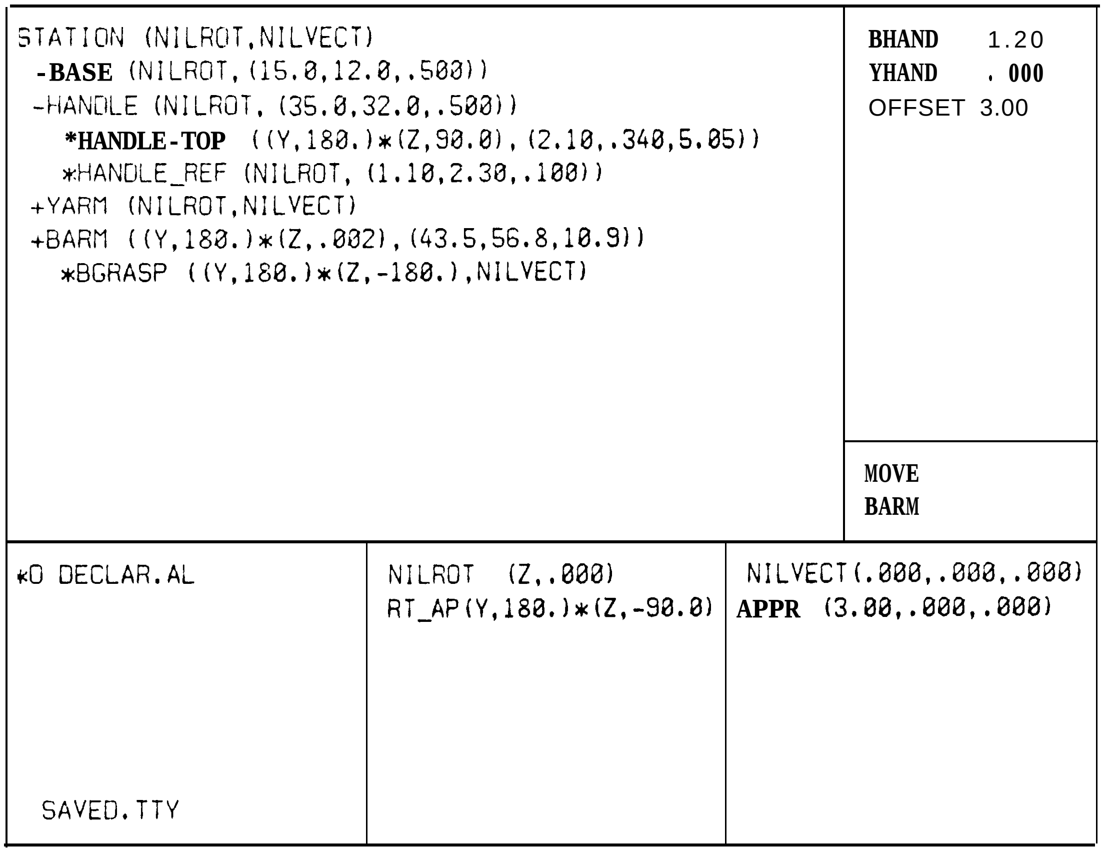
6.2 Executing POINTY

6.2.1 Short form execution instructions and display

The simplest way to execute POINTY is to type the instruction (a list of full commands is given in section 6.2.2).

DO POINTY[PNT, HE]

followed by a carriage return. This instruction first loads the PDP-1 1 with the POINTY runtime system and starts it up so that it is continuously reading the arm joints and printing it out on the VT05 screen, If updating stops on the screen at any time, typing <alt>G at the VT05 keyboard should start it up again. The PDP-10 part of the POINTY system is then loaded and started. POINTY generates a display on the screen which is continuously updated as more instructions are executed. The following shows the state of the display after several instructions.



```
STATION (NILROT,NILVECT)
 -BASE (NILROT,(15.0,12.0,.500))
 -HANDLE (NILROT,(35.0,32.0,.500))
   *HANDLE-TOP ((Y,180.)*(Z,90.0),(2.10,.340,5.05))
   *HANDLE_REF (NILROT,(1.10,2.30,.100))
 +YARM (NILROT,NILVECT)
 +BARM ((Y,180.)*(Z,.002),(43.5,56.8,10.9))
   *BGRASP ((Y,180.)*(Z,-180.),NILVECT)
```

```
BHAND    1.20
YHAND    .000
OFFSET 3.00
```

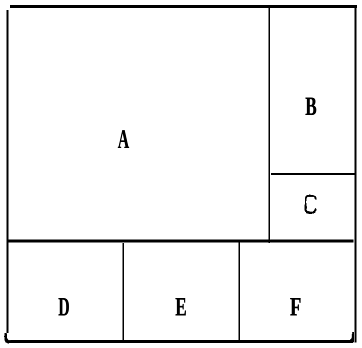
```
MOVE
BARM
```

```
*O DECLAR.AL



  SAVED.TTY
```

```
NILROT  (Z,.000)
RT_AP(Y,180.)*(Z,-90.0)
```

```
NILVECT(.000,.000,.000)
APPR  (3.00,.000,.000)
```

The boxes will be referred to later by the following letters:



A: affixment tree,
  frames and transes
B: scalars
C: default moves
O: output files
E: rotations
F: vectors

The first thing that POINTY wants to know is where to save the terminal session output, by printing out

file for TTY output =<file name> <cr> {filename may be omitted)

A list of ail the POINTY commands given during the terminal session will be saved in <file name> if one is given. Otherwise the terminal output will not be saved. The file name is shown at the bottom of box D. Note that carriage return is the activation character for most instructions, and that the two forms of the AL *COMMENT* statement are valid in POINTY.

POINTY is ready to accept instructions after this exchange, prompting with an asterisk, as it does each time it awaits a new command. Single instructions may terminate with a carriage return or with a semi-colon and a carriage ret urn, and POINTY will then try to execute the instruction. Multiple instructions on the same line must be separated by semi-colons, and the last instruction followed by a carriage return. On seeing a carriage return, POINTY tries to execute the instruction if it is meaningful, otherwise it will await more input and the next carriage return by prompting with ****>>>.

e.g. a 1← 3 <cr> (POINTY will assign value 3 to variable a1}

a l ← 3 + <cr> {POINTY will wait for more input}

in the initial state of the display, Box A indicates the three frames known to POINTY: *station*, *balm*, and *yarm*. The last currently has its coordinates all zero because the yellow arm is disconnected. Box B has values of *bhand* and *yhand* corresponding to the hand opening of the blue and yellow arms respectively. Boxes E and F initially contain the definitions of the predefined rotation *nilrot* and the predefined vector *nilvect*.

6.2.2 Full instructions to run POINTY

The following is a complete sequence of operations required to load and execute t he POINTY runtime system on the PDP-11 and to load and run POINTY on the PDP-10. it is given in case some error occurs when the user does a DO POINTY[PNT,HE].

1. Type “A ELF” to have the ELF (PDP-11 interface) assigned to your job, so that some other job will not try to use it while you are running your program.

2. Type "R11TTY" to execute the program that loads your program into the PDP-11. 11TTY will respond with

```
CORE SIZE = 2SK
VE R SION USING <device>
TYPE ? FOR HELP
*
```

where device is either VT05 or TERMINAL. The asterisk is 11TTY's way of prompting for user input.

The way to change (toggle) between the two devices is to type "V" immediately after the asterisk, and 11TTY fills in the rest of the line and asks for the next prompt as follows;

```
*VERSION USING <other device>
*
```

It is desirable to use the device VT05 so that once execution starts, the joint angles and the other information relating to the arm will appear on the VT05.

3. Type "Z" to zero out the core, followed by a carriage return to confirm the instruction. 11TTY will respond as follows:

```
*ZERO CORE[CONFIRM] 500000<cr>
*
```

4. The POINTY runtime system is then loaded by typing "G" for getting the core image binary file, followed by the name of the file POINTY[PNT,HE] and a carriage return.

```
*GET SAV FILE - POINTY[PNT,HE]<cr>
*
```

5. The next step is to get the program started by typing "S", then "D" followed by a carriage return.

```
*START AT (462452) (D FOR DDT) -D<cr>
 DDT STAATED AT 130000
*
```

6. Now go to the VT05 and after making sure you are ready to push the panic button type

```
*<alt>G
```

You will see continuous scanning on the VT05, and continual updating of the joint angles and other information.

7. You now have to exit from 11TTY by typing X on your terminal.

```
*X
```

8. You will then be in the monitor, and to run POINTY on the PDP-10, type the

following instruction.

```
R POINTY<cr>
```

This will run POINTY and give the display described in the previous section.

6.3 POINTY instruct ions

There are several classes of POINTY instructions.

6.3.1 Assignment statement

In the POINTY assignment statement, as in AL, the expression on the right hand side is evaluated and assigned to the variable on the left hand side. If the variable on the left hand side has not been declared, the assignment statement implicitly declares the variable as having the type of the evaluated expression. An error rnessage will be generated if the variable has been previously declared, and the right hand side expression type is different from the left hand side.

Examples:

s4← 2*3;	declares s4 as a scalar, will appear in box B
v4←zhat+ yhat;	declares v4 as a vector, will appear in box F
r5← nilrot;	declares r5 as a rot, will appear in box E
f5← bpark;	declares f5 as a frame, will appear in box A

6.3.2 Declaration statement

Explicit declarations of *SCALAR, VECTOR, ROT, FRAME* and *TRANS* data types may be made as in AL,

The following AL predeclared variables and constants are recognized by POINTY - scalars: *bhand, yhand*, vectors: *nilvect, xhat, yhat, zhat*, rotation: *nilrot*, frames: *station, barm, yarm, bpark, ypark*, trans: *niltrans*, and dimensional constants; inch, *inches, deg, degree, degrees.* Where *barm* or *yarm* or a frame attached to an arm is used in an expression, the current value computed from the present arm . position will be used.

POINTY allows greater flexibility in specification of explicit data types than AL. in particular, since the number and data types of arguments are different for the various data types (except between *FRAME* and *TRANS)*, they may be declared without the qualifier. if POINTY is unsure whether a declaration is for a frame or a trans, it will assume it is a trans, and will change it to a frame type when the variable associated with it is used in an affixment statement, Note that in the display the reserved words *VECTOR, ROT, TRANS, FRAME* are left out to save space, and that *xhat*, *yhat*, and *zhat* are abbreviated x, *y*, *z*, where it is obvious.

vectors;	VECTOR(<scalar> , <scalar> , <scalar>) or (<scalar> , <scalar> , <scalar>)
rot at ions:	ROT(<vector>, <scalar>) or (<vector>, <scalar>)
frames:	FRAME(<rot>, <vector>) or (<rot>,<vector>)
t ranses;	TRANS(<rot>, <vector>) or (<rot>,<vect or>)

Examples:

valid scalars:	a, a_b, +.01, 3.001
valid vectors:	VECTOR(0, +5, -0.3) (a_b, a , (.05 - a)) (1,+5,.01)
valid rotations:	ROT(xhat, 180) (zhat, 90) (yhat, a>

valid frames:	FRAME(r1*r2, vec1)
	FRAME(ROT(yhat,90),(1, 1, 1))
	(r 1,VECTOR(2.3, a, -.3)

valid t ranses;	TRANS(r1*r2, vec1)
	TRANS(ROT(yhat,90),(1, 1, 1))
	(r 1,VECTOR(2.3, a, -.3)

6.3.3 Deletion statement

Variables may be deleted by means of the delete statement, if the deleted variable is a frame identifier any subtrees rooted in it are also deleted. Examples of the delete statement are

```
DELETE s1,s2,s3,v1,v2,f1,f2;
QDELETE s1,s2,s3,v1,v2,f1,f2;
DELETE ALL;
QDELETE ALL;
```

The use of *ALL* deletes all the user declared variables. If no argument is given, it is assumed to be *ALL,* but in the case of *DELETE,* the user is asked to confirm that he does in fact want to delete all the variables. The variables will disappear from the relevant boxes in the display. If a variable name is given that POINTY does not understand, POINTY will assume a spelling error, and let the user correct the name. If the user does not want POINTY to inform him that the variable does not exist, he should use *QDELETE* instead of *DELETE*. The *QDELETE* command is useful when macros or identifiers are to be read in from a file whose names may be the same as those already defined in the symbol table.

6.3.4 Functions and Macros

A facility for text macro substitution is available, The syntax is similar to that in AL, and defined as shown in the following examples.

```
DEFINE ARM = ⊂barm⊃;
DEFINE V1(A,B,C)=⊂VECTOR(A,B,C)⊃;
```

Note the delimiters used around the body of the macro definition. In the macro definition, the parameter names must be hitherto undeclared variable names. Using those names for some other purpose in the future will not affect the macro definition.

The macro name can be used just about anywhere where the body gives a valid statement or statements. Thus the following are valid:

```
MOVE ARM TO BPARK;
VECT1← V1(0,0,1);
VECT2← V1(⊂2*3⊃,1,4);
```

Note also the use of the delimiters when the parameter substituted is not a single token but an expression or a series of tokens.

Functions may be defined when a complicated expression needs to be evaluated several times. While they may be stored as text macros, a complicated expression is best stored as a function, since the storage is then in an internal form which allows POINTY to check that the expression is valid and has the right operations performed in the arguments, The function definition has the following syntax:

```
<type> FUNCTION <function name> =
            <valid POINTY expression>

<type> FUNCTION <function name>(<type> <argument list>) =
            <valid POINTY expression>
```

```
<type> FUNCTION <function name>(<type> <argument list>;
                    <type><argument list>;.,,
                    <type><argument list>) =
               <valid POINTY expression>
```

Here are examples of the use of functions:

```
SCALAR FUNCTION F1=2*3;
SCALAR FUNCTION SQUARE(SCALAR XI) = X1*X1;
SCALAR FUNCTION DOTPROD(VECTOR V1,V2)=V1.V2;
VECTOR FUNCTION CHANGEXCOMP(VECTOR V1;SCALAR S1) =
        VECTOR(S1,V1.YHAT,V1.ZHAT);
```

Functions are similar to typed procedures in AL, except that the body consists of the expression to the returned, For example, the AL equivalent of the above functions are as follows:

```
SCALAR PROCEDURE F1;
    RETURN{2*3);

SCALAR PROCEDURE SQUARE(SCALAR XI);
    RETURN(X1*X1);

SCALAR PROCEDURE DOTPROD(VECTOR V1,V2);
    RETURN(V1.V2);

VECTOR PROCEDURE CHANGEXCOMP(VECTOR V1;SCALAR S1);
    RETURN{VECTOR{S1,V1.YHAT,V1.ZHAT));
```

Functions may reference user defined identifiers, but in that event the function will be invalidated when the identifier is deleted, The function will be valid when the identifier is redefined.

The current value of a global variable, or an expression of global variables or an arithmetic expression may be evaluated by making use of the EVAL function within the function definition.

Consider the following cases

```
V1← VECTOR{2,0,0)
FUNCTION F1=2*V1
FUNCTION F2 = EVAL(2*V1)
V1 ← VECTOR(0,0,2)
```

A call to F1 now will give the value VECTOR(0,0,4) while a call to F2 will give the value VECTOR(4,0,0).

6.3.5 Expressions

POINTY accepts all the algebraic expressions except boolean expressions that the AL parser is capable of handling, POINTY does not make any checks for dimensional compatibility, The following summarizes the valid operations. Where they have the same meaning as in AL, they are not described in detail.

SCALAR — s+s, s-s, s*s, s/s, s↑s, v.v, |s|, |v|, |r|, s MAX s, s MIN s, s DIV s, s MOD s, INT(s),SIN(s),COS(s),TAN(s), SQRT(s),ASIN(s),ACOS(s),ATAN2(s,s),LOG(s),EXP(s)

VECTOR — s*v, v*s, v/s, v+v, v-v, v*v, r*v, POS(f), f*v, v WRT f, UNIT(v), AXIS(r), t *v,

v REL f ≡ f*v — {*v* is a vector expressed in the coordinate frame *f*. The expression represents the coordinates of the vector in station coordinates.)

ROTATION — ORIENT(f), r*r

FRAME — f+v, f-v, t*f, f*t

f1 REL f2 ≡ f2*f1 — {*f1* is a frame expressed in the coordinate frame *f2*. The expression determines the frame expressed in stat ion coordinates.}

CONSTRUCT(v,v,v),CONSTRUCT(f,f,f) — {constructs a frame using the location part of the three frames, or the three vectors: the first position defines the origin, the second the x-axis and the third the x-y plane of the desired frame. This avoids having to guide the manipulator to a desired orient at ion precisely,}

↑f, ↓f, $f, αf — {returns a frame having the location part as that of f but with different orientations. ↑ gives the vertical component of orient at ion, i.e. if ORIENT(f) = rot(zhat,a) * rot(yhat,b) * rot(zhat,c) then ↑f = FRAME(rot(zhat,c), POS(f)); ↓ gives the orient ation of bpark position, i.e. rot(yhat,180); $ gives the station orientation, i.e. nilrot, and α gives the orientation of bgrasp when the arm is in the park position, i.e. rot(zhat,180)}

TRANS   f→f, t*t, INV(t)

6.3.6 Affixment tree operations

6.3.6.1 *AFFIX & UNFIX*

The *AFFIX* instruction is similar to the AL *AFFIX* instruction, but allows *RIGIDLY* and *NONRIGIDLY* to be abbreviated as "*" and "+" respectively;

```
AFFIX f 1 TO f2;
AFFIX fl TO f2 RIGIDLY;
AFFIX f 1 TO f2 NONRIGIDLY;
AFFIX f 1 TO f2 *;
AFFIX fl TO f2 +;
```

Frame *f1* is affixed to the *f2*. Unless specified otherwise, the affixment is *RIGID*. Every newly defined frame is shown with respect to the *station* frame (indicated with a "-" on the display). The affixment trees appear on box A of the display as they are constructed. Frames may also be affixed with the relative transform between them being specified as follows;

```
AFFIX <identifier> TO f3 AT (<rot>,<vector>);
AFFIX <identifier> TO f3 AT (<rot>,<vector>) RIGIDLY ;
AFFIX <identifier> TO f3 AT TRANS (<rot>,<vector>) NONRIGIDLY ;
```

If <identifier> is not a frame, a new frame is defined before it is affixed. This instruction is used mainly for reading in AL instructions generated during a previous POINTY session.

The *UNFIX* instruction is written as in AL or in a short form as shown below. *Frame_1* is unfixed from *frame_2* and affixed independently to *station*.

```
UNFIX frame-l ;
UNFIX frame-1 FROM frame,2;
```

6.3.6.2 *COPY*

The *COPY* instruction is used to affix a copy of a frame and its associated affixment subtree to another frame. The syntax is as follows:

```
COPY <frame_1> INTO <frame_2>;
```

POINTY prefixes the first part of the name of <frame_2> (the part before the underscore, if there is an underscore, or the full name, if its length is less than 5 characters, otherwise the first three characters) or a user defined prefix to the frames in the subtree of <frame_1>. Any frames in the subtree of <frame_1> having an underscore will have the part after the underscore suffixed to the prefix determined from <frame_2>. if the procedure results in duplicated frame names, POINTY allows the user to specify a new name.

A copy of only the subtrees of one frame to another may be made by using

COPY SUBTREE(<frame_1>) INTO <frame_2>;

Examples are given of the use of the COPY command to two affixment trees rooted in *base* and *handle* as shown.

```
station (nilrot,nilvect)
 -base (nilrot,(15.0,12.0,.500))
 -handle (nilrot,(35.0,32.0,.500))
   *handle_top ((Y,180.)*(Z,90.),(2.10,.340,5.05))
   *handle-ref (nilrot,(1.10,2.30,.100))
```

The instruction *COPY SUBTREE(handle) INTO base* produces the following result:

```
station (nilrot,nilvect)
 -base (nilrot,(15.0,12.0,.500))
   *base-top ((Y,180.)*(Z,90.0),(2.10,.340,5.05))
   *base_ref(nilrot,(1.10,2.30,. 100))
 -handle (nilrot,(35.0,32.0,.500))
   *handle-top ((Y,180.)*(Z,90.0),(2.10,.340,5.05))
   *handle_ref (nilrot,(1.10,2.30,.100))
```

The names of new frames are obtained with the previously explained convention: the name of the “receiving” frame, *base*, is taken as the prefix for the new names and it is substituted for the part of the names before the underscore. Ail the sons of the frame handle are copied as sons of the frame *base*.

The instruction *COPY handle INTO base* will produce this result.

```
station (nilrot,nilvect)
 -base (nilrot,(15.0,12.0,.500))
   *base_handle (nilrot,(35.0,32.0,.500))
     *base_top ((Y,180.)*(Z,90.0),(2.10,.340,5.05))
     *base_ref (nilrot,(1.10,2.30,.1 00))
 -handle (nilrot,(35.0,32.0,.500))
   *handle-top ((Y,180.)*(Z,90.0),(2.10,.340,5.05))
   *handle_ref(nilrot,( 1.10,2.30,.100))
```

Handle together with its subtrees, is copied as a son of the frame base. The convention used for producing new names generates *base,handle,* since the name *handle* has no underscore.

6.3.7 Arm interact ion commands

The interface to the POP-1 1 is used for arm interactions. The current position of the arm is passed back to the PDP-10, while instructions to tell the arm to perform some motion are sent to the POP-11. When arm interaction occurs, a software interlock prevents execution of the next POINTY instruction until the arm position is read, or the motion is completed or determined to be unsuccessful.

6.3.7.1 Arm reading commands

Arm positions are read directly each time an expression is evaluated. The current arm position is used whenever the arm is referred to directly, and to compute the values of any affixed frames. The user may not assign values to the frames *barm* and *yarm*.

Two particular assignments are required to initialize the system. The arm is moved to a *FIDUCIAL* point (an arbitrary reference point whose location is defined by this statement) and the following command is given:

FIDUCIAL ← <arm>;

To find the position of the pointer, the pointer is grasped in the arm and used to point to the fiducial point. The following command is given:

POINTER ← FIDUCIAL;

An *AFFIX* statement is then used to affix the pointer rigidly to the relevant arm.

The *FCONSTRUCT* command is used to construct a frame from three readings. A sample call is as follows:

FCONSTRUCT f;

where f is an undeclared identifier, POINTY asks which device (*barm*, *yarm* or *pointer*) to use and lets the user specify the meaning of three posit ions by pointing the manipulator at them. The first is always the origin of the desired frame, the second on one of the principal axes (*xhat*, *yhat* or *zhat*) specified by the user, and the third on the plane determined by that axis and another of the remaining principal axes specified by the user. The three locations are used to compute the desired frame.

6.3.7.2 *MOVE* command

The arm motions specified can be absolute or differential. The syntax of absolute moves is similar to the basic ***MOVE*** instruction in AL.

```
MOVE f1 TO <fr_exp>
```

This is the general move command, *f1* (assuming it is affixed to an arm) is moved to the indicated destination. The destination of the movement can be specified in terms of the location of *f1* at the start of the motion.

POINTY does not currently know about approach and departure points, deproaches, force sensing, condition monitors, ***or*** VIA points. However, a series of single segment moves may be specified by giving a a list of frame expressions instead of a single frame expression, as follows:

```
MOVE f 1 TO <fr_exp_1>,<fr_exp_2>,...,<fr_exp_n>
```

where the maximum number of expressions is 9.

Differential moves (not directly possible in AL but achievable by defining a macro which expands to cTO⊗+⊃) can be specified by using a *BY* instead of *TO* and a vector instead of a frame expression, as follows:

```
MOVE f 1 BY <vector>
MOVE f 1 BY <vector> WRT f 2
```

These instructions are equivalent to:

```
MOVE f 1 TO ⊗ + <vector>
MOVE fl TO ⊗+ <vector> WRT f2
```

Differential moves parallel to the x, y or z axes of the stat ion may be specified by the following instructions.

```
MOVEX f 1 BY <scalar>;
MOVEY f 1 BY <scalar>;
MOVEZ fl BY <scalar>;
```

These instructions are equivalent to the AL instruction

```
MOVE f 1 TO ⊗ + <scalar>*<axis>
```

To reduce repetitive typing, a move instruction similar to the last executed move instruction (shown in box C of the display) may be given by merely typing the last part of the instruction. Hence it is possible to state

```
TO fl;
TO f 1 + <vector>;
TO f 1 + <vector> WRT f2;
BY <vector>;
BY <vector> WRT f2;
BY <scalar>;
```

the last form may be used only after a differential movement instruction along a principal axis *xhat*, *yhat*, or *zhat*.

Since movement to the park position is done so often, it may be abbreviated as follows:

```
PARK BARM;          { same as MOVE BARM TO BPARK; }
PARK YARM;          { same as MOVE YARM TO YPARK; }
PARK;               { parks both arms }
```

6.3.7.3 *CENTER* command

The syntax and use of *CENTER* is similar to that in AL.

```
CENTER <arm>; {<arm> may be left out }
```

closes the fingers slowly, moving the <arm> to accomodate to the location of any object positioned between the fingers. If <arm> is left out, the last arm moved will be used.

6.3.7.4 ***OPEN*** and *CLOSE* commands

The syntax for hand motions are similar to those in AL except that differential movements may also be specified.

```
OPEN <hand> TO <scalar>;
CLOSE <hand> TO <scalar>;     { absolute opening or closing }
OPEN <hand> BY <scalar>;
CLOSE <hand> BY <scalar>;     { differential opening or closing }
```

If the next motion statement is to open or close the same hand, the instruction may be abbreviated as follows:

```
      TO <scalar>;
or    BY <scalar>;
```

6.3.7.5 *DRIVE* command

POINTY permits the movement of individual joints (which is not permitted by AL). The syntax is as follows;

```
DRIVE BJT(<joint number>) TO <scalar>;
DRIVE YJT(<joint number>) TO <scalar>;
DRIVE BJT(<joint number>) BY <scalar>;
DRIVE YJT(<joint number>) BY <scalar>;
```

The indicated joint of *barm* or *yarm* is moved to <scalar> or by <scalar>. <Joint number> is an integer which represents the joint; joint 7 is the hand, while joints 1 through 6 are the arm joints. Driving joint 7 is equivalent to the *OPEN* or *CLOSE* instruction. <Scalar> represents angles in degrees for joints 1,2,4,5,6 and displacement in inches for the prismatic joint 3 and the hand joint 7.

Short forms exist as for the other motion instructions.

```
      TO <scalar>;
or    BY <scalar>;
```

6.3.8 Display routines

The standard display has been described in section 6.2.1, and it shows as much useful information as possible by omitting the use of reserved words like *VECTOR*, *TRANS*, etc, and by abbreviating *XHAT*,*YHAT*,*ZHAT* *to* *X*, *Y*, *Z*, and not displaying the values of POINTY defined constants. A movable arrow is available to highlight frame variables of interest.

There are now three display modes available. POINTY is initialized in the table display mode, in which scalars, vectors, transes, frames, rots, the default move statement, and the files used in the current session are shown. Owing to lack of space on the display, macros and function expression definitions are not displayed in this mode. The type display mode allows the user to see all the current definitions of the specified data type.

```
DISPLAY <data type>{ where <data type> is SCALAR, VECTOR,
                         ROT,TRANS,FRAME,
                         MACRO,FUNCTION }
```

This display mode permits the display of more variables of a data type than is possible in the standard display, and the display of macros and functions. When the user is more interested in seeing what he has typed so far, the display mode most useful is the no display mode, invoked by the command

```
NODISPLAY
```

This eliminates the display altogether, and just prints out the series of commands typed by the user, To get back to the table display mode, the redisplay command needs to be used.

```
REDISPLAY
```

6.3.9 File input/output

File input and output is necessary to generate the affixment trees for AL instructions, as well as to save the results of a POINTY session and to make use of the results of previous sessions.

6.3.9.1 Saving current state -*WRITE. CLOSE*

The *WRITE* instruction is used to write on the indicated file the AL instructions required (declarations, assignments, and affixments) to define variables and preserve the current state of the world. The syntax is as follows;

```
WRITE;                      { into file last written }
WRITE <id>;                 { into file last written }
WRITE INTO <file>;          { write everything into <file> }
WRITE <id> INTO <file>;
```

If the *<id>* part is omit ted, all the variables (except station, *fiducial, pointer, yarm* and *barm* and other predeclared variables) are output, otherwise only the indicated frame and the subtrees rooted in it, or the identifier is output. Since frames are affixed to other frames in terms of their relative transes, any frame to be saved should be affixed independently to *station* in order to obtain its absolute location.

POINTY permits output to different files. If the file named does not exist, it is created, and the current time and date written out before the required information specified by the user. If it exists and is open and previous output has been done to the file but a *CLOSE* not done on it, the output is appended. If it exists but no input has been done to it, the current time and date are put on a fresh page, followed by the desired output. If no output has been done so far, or no new output has been performed since the last *CLOSE*, and no file name is specified, output will be directed to a file DECLAR.AL on the area of the current user. Otherwise the last file written is used.

Files should be closed if there is no further output to be directed to them, and before ending a POINTY session. The command syntax is as follows:

```
CLOSE;                 { closes last file written }
```

CLOSE <file>;	{ closes file <file>; future output will go onto the next page }
CLOSE-FILES;	{ closes all files currently opened, including the one collecting the terminal output, after asking user for confirmation, after which POINTY asks for a new file name in which to collect future terminal output }

To ensure that this CLOSE instruction does not interfere with closing a hand, the user should try not to use a file called BHAND or YHAND. All files will be closed at normal exit for POINTY if they have not been closed explicitly by the user.

6.3.9.2 Getting a given world state - *REAL, & QREAD*

The *READ* and *QREAD* commands will read the specified file of AL instructions to bring the state of POINTY's world to a known state, or to a state that was saved at the end of the previous terminal session, so that in addition to being input files to AL, POINTY generated files may be used to store instructions to build the necessary frame tree structure and assign values to variables.

```
READ;             { reads from DECLAR.AL }
READ <file>;
QREAD;
QREAD <file>;
```

Since movement commands may also be given in the input file, the user should be careful that the commands do not cause disastrous motions to occur. The *READ* command will print out the input file as it is being read. The *QREAD* command will execute faster since it does not print out the input file.

6.3.10 Miscellaneous commands

```
EDIT <variable>;
```

loads the line editor with the value of the variable and allows the user to edit it. This is particularly useful when the user wants to change the rotation part of a transformation without changing the vector part. It can also be used to change the definitions of macros and functions.

```
PRINT <expression>;
```

prints out the value of the arithmetic expression.

```
SPRINT "<any text >";
```

prints out <any text> at the user terminal

PROMPT

waits for the user to type "P", followed by a carriage return before proceeding any further.

RENAME <variable>;

allows the user to change the name of the variable.

EXIT;

exits from POINTY.

Typing a question mark

?

gives some information to the user about the available instructions and their meaning. Whenever a syntax error is detected this instruction allows the user to obtain information about the correct syntax of the statement,

↑, ↓, n↑, or n↓

shifts the display arrow up or down. n determines the distance the arrow is to be shifted.

Some error recovery procedures are available, Whenever an undeclared identifier is used where POINTY expects a known variable or its value, POINTY will keep asking for a corrected name until it is given something it can work with, or the user hits <control>C to get out of the query loop.

<ESCAPE>I

typed on the terminal will cause termination of program execution at the end of the current input line or statement, whichever comes first. All typeahead will be flushed, and if POINTY is reading from an input file, it will stop. The next input accepted will be from the keyboard. This command is used to get out of runaway executions when instructions are being executed from a file through the *READ* command, or out of infinite loops,

6.4 Hints on using POINTY

6.4.1 Recommended seauence

The following is a recommended sequence of steps for using POINTY after initializing it:

1) Use the arm to grasp the fiducial point and type the instruction

FIDUCIAL ← BARM;

2) Put the pointer in the hand of the arm and grasp it tightly. Point the tip of the pointer to the fiducial point and type the instruction

POINTER ← FIDUCIAL;

3) Now affix the pointer to the arm frame

AFFIX POINTER TO BARM;

4) For any object, it is desirable to find a reference point for the reference frame. In order to be able to locate the object quickly for future use, it is desirable to have the orient at ion parallel to the station orient at ion. Thus the pointer should be used to point at the reference frame, and the following instruct ion typed

origin ← 8 POINTER;

5) The frames for other features of interest are found by using *barm*,*pointer*, *CONSTRUCT* or *FCONSTRUCT*. Let us call these new frames *f1*, *f2*, *f3*;

6) These frames should be affixed to the origin by the instruct ions

AFFIX fl TO origin RIGIDLY;
AFFIX f2 TO origin RIGIDLY;
AFFIX f3 TO origin RIGIDLY;

7) Before exiting from POINTY, do not forget to save the frame tree you are interested in.

6.42 Hints

1) It is possible to record the values of variables during a session by asking to edit those variables. The values will be saved within the file collecting the terminal output.

2) If the POINTER is physically removed from the arm, the user need not bother to *UNFIX* it. So far as POINTY is concerned, there is an imaginary pointer in the hand. If the user can put back the pointer in the same position later, the values will still be valid. For access to difficult places, the bendy pointer (6.1.2.2) can be used; however, it must be redefined each time it is bent.

3) Certain positions may be read more easily by moving the arm there and grasping, rat her than using the pointer. In that case the value of barm should be used.

4) It is a good idea to save the current value of a frame within another variable before moving it, so that if you later decide to backup, the value will be available.

5) While objects may be in any arbitrary orientation, it is generally easier to use POINTY if the principal axes of frames are parallel or orthogonal to the stat ion axes.

7. ERROR CORRECTIONS AND RECOVERY

Errors can occur at various stages during program compilation and execution, and it is important to be able to continue from the error point as gracefully as possible. Some errors may be patched up according to the wishes of the user, while others may be fixed up by the AL system, with the user having no say other than whether to continue with the execution or to abort it. This chapter attempts to describe the kinds of errors encountered during program compilation and execution, and what action the user can take when such errors do occur.

7.1 Parsing errors

Errors detected in the parsing phase are the easiest to correct and patch. For minor errors it is possible to proceed after correction without going back to the source file.

The parser outputs error messages, and gives the user the option of

(a) editing the source file
(b) aborting the compilation
(c) taking the standard fixup
(d) backing up to and changing the source code from the beginning of the innermost statement.

The last feature is particularly advantageous when the compilation is a long one, and the error is a minor one which can be easily corrected - e.g. errors which are due to misspellings, missing operators, and even some simple cases of syntactically incorrect statements.

Error messages are generated whenever the parser comes across something it does not like. Some messages are warning messages which tell the user what he should not do in the future, An example of this is the case where identifiers are declared in a block but never referenced, resulting in carrying more variables than necessary, or where an identifier is not expected to have a planning value, thereby causing problems in the compilation phase.

The most common errors are dimension and type incompatibility. Dimension checking is done across assignment statements and force, torque, and duration expressions and conditions, Whenever there is inconsistency, an error message is generated. While dimensional inconsistency may not cause any grief during execution of the program, checks for it enable certain errors (e.g. wrong variables being used) to be pinpointed early during the compilation phase. A more serious error occurs when the data types are incompatible (e.g. assigning a vector expression to a scalar variable), and needs to be corrected, as otherwise the error will cause trouble in the compilation and execution phases.

Dimension checking can be made less stringent by including the F switch in the REQUIRE ERROR-MODES statement, In this case, dimensionless variables will be coerced to the type that makes them compatible with the other terms in the expression or stat ernent.

TYPICAL ERROR MESSAGES;

TYPE MISMATCH

This message is printed when an identifier, factor or term is of a different type than that expected in the context of the expression.

DISTANCE DIMENSIONS DON'T MATCH ON ASSIGNMENT STATEMENT

The meaning is obvious, but the error is not serious and AL will allow the code to continue compiling, since dimension checking is not done during execution of the program, --

BLOCK NAME AT END DOES NOT AGREE WITH THAT AT THE BEGINNING

This error occurs when there is a misspelling in the names within strings at the corresponding places, or if the *BEGINS* and *ENDS* are mismatched.

TRYING TO ASSIGN VALUE TO ARM OR DEVICE

The user is trying to assign *a* value to an arm or a hand. This is disallowed in a program because the values reflect the state of the real world during execution, and cannot be changed by the user,

-*<variable> NOT DEFINED, WILL DEFINE IT.*

The user has put an undeclared variable on the left hand side of an assignment statement. This message could be due to a misspelling.

An error message, followed by *CONTINUE WILL FLUSH STATEMENT*

This means that the parser will be unable to do any form of fixup, and that it will just flush the statement by ignoring any further text until the next semi-colon is read.

ERROR CORRECTION

Whenever the parser detects an error, it prompts the user with a "$". The user should respond with a single character as follows:

C or <cr>	continue with standard fixup
A or <lf>	continue automatically with standard fixup for future errors.
E	edit source file at the place the error occurs
R	restart the program - type in the command line
T	terse information giving only the different options available
V	verbose information giving characters and their effects
X	exit from program
L	log errors in a logging file
M	modify source code - user will be presented the offending line and given the chance to modify it

Any other character will cause either a list of the above information to be printed out, or just a list of the possible options.

The most useful response for the user is "M", "C" or "E". The first is particularly useful when a minor error (e.g. spelling error) occurs towards the end of a long compilation, and the user does not want to have to start from the beginning again. "C" is useful when the error can be corrected by a standard fixup, while "E" is used to correct more serious problems by going back to the source file,

Note that the "M" option is not always available. There are situations where interactive error recovery is impossible - e.g. when in the middle of a macro expansion, and so the user is not allowed to make any changes.

If any errors have been corrected interactively, at the end of the parsing phase AL will ask the user if an updated copy of his source file is to be saved.

7.2 Compiler errors

The principal compiler error messages are those given when planning values are not available to variables, (e.g. at the beginning of a procedure its parameters are undefined) or the arm is expected to move to some impossible position, e.g. below the surface of the table, or a position corresponding to joint values beyond the joint stop limits.

Some messages are WARNING messages, which means that compilation wilt cont inue automatically. Some others are HAH! messages, which may ask for user response by means of an up arrow "↑". In such cases the user usually has few options other than to continue with the standard fixup by typing C or to abort the compilation and change the source program, Note that once the parser exits to ALC, the user has no more cont rol over changing the contents of his original source code without running the parser again or changing the S-expression file.

TYPJCAL MESSAGES:

WARNING: <variable> HAS NO PLAN VALUE -WILL USE ZERO.

When doing the world modelling (see section 4.6), if the plan tirne value of a variable is needed, but it has not been assigned one this message is printed with *<variable>* equal to the name of the undefined variable. A plan time value of zero, nilvect, nilrot or niltrans will be used depending on the data type of the variable.

WARNING: BLUE ARM NOT PARKED UPON PROGRAM COMPLETION.

It is good practice to park the arm before exiting from the program.

HAH! You want only 2.000000 for this motion, and I think you need 2.135642 . In order to satisfy your request, 1 am disregarding any other time constraints you. may have placed on the motion.
Called from 544264 Last SAIL call at 536712

This message is given when the user asks for a motion duration time which is too short.

JOINTS OUT OF RANGE: . . .
NAN! This destination location is not accessible.
The closest reasonable point is bsing used.
Called from 544264 Last SAIL call at 536712

The message here is obvious. It means that there is some mistake in what the user is asking for - maybe trying to reach below the table, or trying to get to a location which requires the joints to go beyond their limits, If you think that the location is accessible, it may be that the desired location is high above the location of It he table top with the hand pointing downward vertically, and that would not be possible because joint 5 will run into a stop limit. This problem may occur if the hand has the park orientation at a height more than 10 inches from the table top.

7.3 PALX errors

The principal PALX error occurs when the program is very long. PALX then gives the message that there might not be enough space, in which case, the program should be broken down into smaller subprograms. If any other error message is given by PALX, it is an AL bug, and the user is requested to report it.

7.4 Loading errors

11 TTY is the program that loads the PDP-11 with the core image of the AL interpreter and runtime system. The instruction "DO AL[AL,HE]" has the effect of a number of instructions which includes assigning the PDP- 11 to your job, zeroing the memory of the PDP-11, loading the AL interpreter and runtime system, overlaying it with the user program load module and starting DDT on the PDP- 11. Further details are given in Chapter 5.

There are several things that could go wrong during this sequence of events, The message will be printed out at the terminal of the user,

ELF ASSIGNED TO JOB n.

This message is printed when some other job has the ELF (PDP- 11 interface) assigned to it. When this happens, you should find out whose job has it assigned, and see if the owner is using it, If the job is not using the ELF, you should request that the ELF be deassigned, and then try the "DO" instruction again. If the job is using the ELF, you should try again later.

PDP-II STOPPED, RESTART

Restarting the entire sequence from Zeroing the core will take care of this problem.

NO RESPONSE WHEN YOU TYPE ANYTHING ON THE VT05.

If there is no response on the VT05 when you expect some output e.g. when you do not get the asterisk and the flashing cursor, 11TTY may be in "'TERMINAL" rather than "VT05" mode, Type V several times on the terminal and let the mode toggle from one to the other until "VT05" mode is obtained,

7.5 Runtime errors

During the execution of the user program, several things can cause the program to stop. The following are the common error messages that are printed on the VT05 by the runtime interpreter.

INCOMPATIBLE PCODE VERSION. PROCEED AT YOUR OWN RISK

This means that the binary file assembled for the user program is incompatible with the current runtime system. The solution is to recompile the user AL program.

FREE STORAGE EXHAUSTED

Only very large programs will cause this error. It has been largely eliminated with the addition of more memory,

NO VALUE FOR VARIABLE - USING DEFAULT.

This error is caused by at tempting to access a variable before it has been assigned a value. Proceeding will use a value of zero, nilvect, nilrot, or niltrans, depending on the data type of the variable.

USER PDLOV

This is a fatal error caused by a bug in either the hardware or the runtime system. Sometimes restarting the program will cause this error to go away,

Below are the errors associated with the arms that occur during motions.

CAN'T INITIALIZE ARM. REFERENCE POWER SUPPLY OUT OF RANGE.

The arm initialization routine ran into trouble due to the arm reference power supply drifting. The program may be continued by typing <alt>P, but this should be done with extreme caution, and the user should be extremely alert with a finger over the panic button to cause an immediate stop if the arm does somet hing unexpected.

PAN ICBUTTONPUSHE D

This error occurs when the panic button is pushed, or someone has leaned on the edge of the table, thereby pulling on the yellow cord, and shutting off the power supply. RETRY<alt>G will try the current motion again if the panic butt on was pushed, but it will give the next message if the yellow cord was pulled.

ARM INTERFACE POWER SUPPLY TURNED OFF
(CHECK JOINT BRAKE SWITCHES)

When this error message appears, check all the brake switches on the panic button box, and make sure that all the brakes are applied. If any of the brakes are in the released position, toggle them to the set position, and then try again by typing RETRY<alt>G. if you get the same error again, press the large red button on the underside of the short side of the table nearest the wall to turn on the arm power, and try again. If you get the error again, it may be that the arm interface power really is off at the source, in which case you should get help from one of the personnel in the lab.

EXCESSIVE FORCE ENCOUNTERED BY JOINT n

This error occurs when the movement to be made requires too high a force. It could occur:

(1) when the arm encounters an object during the course of the motion (get it out of the way)

or (2) the time specified for the motion is too short (make it longer next tirne)

or (3) if the position at the beginning of the motion is far from the planned position (user can't do much about it at runtime except move the arm manually where it is supposed to go)

The nurnber at the end will tell which joint ran into problems. Numbers 1 through 7 represent joints 1 through 6 and the hand of the yellow arm, while numbers 10 through 16 represent corresponding joints in the blue arm.

TIME OUT FOR JOINT n

This occurs usually at the end of a motion when the arm is prevented from going to its final destination but the error is insufficient to cause a high enough mot or torque requirement to give a joint force error,

STOP LIMIT EXCEEDED FOR JOINT n

There is a software joint operating range which is lower than the hardware joint operating range for safety purposes, and when the limits are exceeded, this error message is generated. Usually this message occurs if continuation of compilation had been allowed in the compilation phase when a "destination location not accessible" message was generated. Again the offending joint number is indicated,

CONTINUATION FROM ERRORS

To continue from an error there are several possibilities that are indicated . on the VT05.

<alt>G	will cause execution to begin from the start of the program.
<alt>P	will cause execution to continue from the next statement.
RETRY<alt>G	will attempt to retry the aborted move. Note that motions involving force will currently not use the force system in the retried motion.

OTHER ERRORS

Jhe following are internal or hardware errors over which the user has little control. They are given for the sake of completeness. Such error messages should be reported. Many of them will leave the error number in RO in which case typing "RO[" to DDT will print it out.

1	COULD NOT ATTACH TO REQUESTED JOINT(S)
2	INCORRECT NUMBER OF JOINTS REQUESTED TO BE DRIVEN
3	WIPERS COULD NOT BE READ WITHIN THEIR OPERATING RANGE
4	ARM SOLUTION DOES NOT EXIST
5	UNKNOWN TOUCH SENSOR REQUESTED
6	NO MORE FREE SLOTS IN TOUCH SENSOR EVENT LIST
11	ZERO VELOCITY TACHOMETER READING OUT OF RANGE
12	ATTEMPTED TO SWITCH ARMS WHILE FORCE SERVOING
13	NO MORE FREE SLOTS IN FORCE SENSOR EVENT LIST
14	NEED ALL 6 ARM JOINTS IN ORDER TO DO FORCE SENSING/COMPLIANCE
15	CAN'T FORCE SERVO MOTION WITHOUT POLYNOMIAL
20	JOINT STARTED OUTSIDE OF PERMITTED OPERATING RANGE
400	JOINT IS DOWN, INOPERABLE
1000	CATASTROPHIC A/D ERROR HAS OCCURRED
40000	NO ARM SOLUTION WHILE DOING FORCE COMPLIANCE

7.6 Hints on debugging

There are several instructions that are available for the user to determine which part of his program is giving him problems.

7.6.1 Parse time debugging aids

REQUIRE MESSAGE COMMAND (cf. section 4.5.9)

The message can be used to inform the user where he is in the program, but since the user is normally familiar with his program, it would be used where there are long compilations of several source files, and the user wants some description of the contents of some source file. Another use is to output a message to set parameters during compilation, and follow it directly with a REQUIRE SOURCE-FILE "TTY:FOO.AL". The user can then make the required assignments from the teletype.

REQUIRE ERROR-MODES "LA" (cf. sect ion 4.5.9)

This message is particularly useful if the compilation is to be done non-interactively, Errors (if any) will automatically be collected in a file with extension LOG, and the parser will try to continue from errors the best it can.

7.6.2 Compile time debugging aids

NOTE, NOTE1, NOTE2 COMMANDS (cf. section 4.5.10)

If you get messages from the AL compiler that you do not understand, one way to determine where the cause lies in the source program is to use the *NOTE* command as described before. By examining where the error message occurs with respect to the notes printed out by the program, it is possible to localize the source of error.

DUMP COMMAND (cf. section 4.5.10)

This prints out the plan-time value of the variables of interest to allow the user to verify that they are what he expects them to be.

7.6.3 Runtime debugging aids

7.6.3.1 ALAI0

ALAI0 is the high level debugger for AL programs that resides on both the POP-I 0 and POP- 11, providing a communication link between the two computers that permits the user to examine and change the values of variables and allow events to be signalled and waited.

ALAID is run after the AL program being debugged has been loaded and started on the PDP-11 in the usual way.

```
RU DEBAL[AL,HE]<cr>
```

results in running a driver program on the POP-10 that communicates with the ALAI0 process which is running on the POP-I 1 in parallel with the AL runtime interpreter. OEBAL responds with a colon-asterisk-colon whenever it expects a I user command.

```
:*:
```

The symbol table for the variables in the user AL program must be given to ALAI0 by means of the following command:

```
SYMBOL FOO.ALS<cr>
```

where FOO.ALS is the name of the user's AL program symbol table as generated by the AL compiler. When ALAI0 has successfully read in the file {and when it successfully performs any commands given to it) it will type:

DONE

It is possible to use ALAI0 to examine and modify the value associated with the variables in the user's AL program by means of the following two commands:

```
GETVAL (NAME v)
SETVAL (NAME v)(<data type> <value>)
```

where *v* is the name of the variable being refered to, <*data type*> is the data type of the variable (e.g. *SCALAR, VECTOR, FRAME*), and <*value*> is the new value to be assigned.

It is also possible to signal and wait for events with ALAIO. The syntax of these commands is as follows:

```
SIGNAL (NAME e)
WAIT (NAME e)
```

where *e* is the event being used.

A running AL program may be halted and execution continued by means of the *HALT* and *GO* commands as follows:

```
HALT
GO
```

The *HALT* command halts execution of *the* AL program upon completion of execut ion of the current pcode instruction. ALAI0 sends the message "ALL ACTIVE INTERPRETERS HALTED" to the user terminal,

The current version of ALAI0 is primarily used as an interface between the AL runtime system and some higher level program (e.g. a vision module) that is running on the POP-I 0. A more advanced version of ALAI0 is currently being implemented which allows AL wizards to alter the flow of control, set breakpoints, and examine/modify the pcode.

7.6.3.2 *PRINT* statement

A second way to help debug the program during execution is to output values and messages during the execution by means of the *PRINT* command (c.f. section 4.5.7). It is useful for printing out actual values of variables at execution,

7.6.3.3 1 1 DOT

11 DOT is an assembly-language symbolic debugger for the POP-l 1, and its use is outside the scope of this document, It is primarily used by AL wizards to debug the runtime system, The user is exposed to 11 DDT to the extent that he uses it to start or continue execution of his program using the <alt>G, RETRY<al t>G, and <alt>P commands.

Appendix I. AL RESERVED WORDS, PREOECLAREO CONSTANTS AND MACROS

Reserved Words

ABORT
ABOUT
ACOS
AFFIX
ALONG
ALSO
AND
ANGLE
ANGULAR_VELOCITY
APPROACH
ARRAY
ASIN
AT
ATAN2
AXIS
BEGIN
BY
CASE
CENTER
CLOSE
COBEGIN
COEND
COMMENT
COMPILEA-SWITCHES
CONSTRUCT
cos
DEFER
DEFINE
DEPARTURE
DEPROACH
DIMENSION
DIMENSIONLESS
DISABLE
DISTANCE
DIV
DO
DUMP
DURATION
ELSE
ENABLE
END
EQV
ERROR-MODES
EVENT
EXP
FIXED
FOR
FORCE
FORCE-FRAME
FRAME
FROM
HAND
IF
IN
INT
INV
INSCALAR
LABEL
LOG
MAX
MESSAGE
MIN
MOD
MOVE
NO_NULLING
NONRIGIDLY
NOT
NOTE
NOTE 1
NOTE2
NULLING
ON
OPEN
OPERATE
OR
ORIENT
PAUSE
POS
PRINT
PROCEDURE
PROMPT
QUERY
REFERENCE
REQUIRE
RIGIDLY
ROT
SCALAR
SIGNAL
SIN
SOURCE-FILE
SPEED-FACTOR
SQRT
STEP
STOP
TAN
THEN
TIME
TO
TORQUE
TRANS
UNFIX
UNIT
UNTIL
VALUE
VECTOR
VELOCITY
VIA
WAIT
WHERE
WHILE
WITH
WOBBLE
WORLD
WRT
XOR

Predefined constants

π
BPARK
CM
CRLF
DEG
DEGREES
FALSE
GM
INCH
INCHES
LBS
OUNCES
oz
NILDEPROACH
NILROT
NILTRANS
NILVECT
Pi
RADIANS
SEC
SECONDS
STATION
TRUE
XHAT
YHAT
YPARK
ZHAT

Predefined identifiers

BARM
BHAND
YARM
YHAND

Predefined macros

APPROXIMATELY
CAUTIOUS
CAUTIOUSLY
DIRECTLY
PRECISELY
SLOW
SLOWLY

Appendix II. POINTY RESERVED WORDS & PREDEFINED CONSTANTS

Reserved words

ACOS
ALL
ASIN
ATAN2
AXIS
BPARK
BJT
BY
CENTER
CLOSE
CLOSE-FILES
COMMENT
CONSTRUCT
COPY
COS
DEFINE
DELETE
DISPLAY
DISTANCE
DIV
DRIVE
EDIT
EVAL
EXIT
EXP
FCONSTRUCT
FRAME
FROM
FUNCTION
INT
INTO
LOG
MAX
MIN
MOD
MOVE
MOVEX
MOVEY
MOVEZ
NODISPLAY
OPEN
ORIENT
PARK
POS
PRINT
PROMPT
PWRITE
QOELETE
QREAD
READ
REDISPLAY
REL
RENAME
ROT
SCALAR
SIN
SPRINT
SQRT
SUBTREE
TAN
TO
TRANS
UNFIX
UNIT
VECTOR
WRITE
WRT
YJT
YPARK

Predefined Identifiers

BARM
BGRASP
BHAND
BPARK
POINTER
STATION
YARM
YHAND

Predefined Constants

BPARK
DEG
DEGREE
DEGREES
INCH
INCHES
NILROT
NILTRANS
NILVECT
PI
XHAT
YHAT
YPARK
ZHAT

Appendix III. AL COMMAND SUMMARY

```
BLOCKS:          BEGIN S; S; S; . . . S END
                 COBEGIN S; S; S; ... S COEND

DECLARATIONS:    TIME SCALAR ts1,ts2;             LABEL l1,l2;
                 DISTANCE VECTOR dv1,dv2;
                 ROT r1,r2;                       FRAME f1,f2;
                 TRANS t1,t2;                     EVENT e1,e2;
                 FRAME ARRAY f1[s1:s2],f2[s3:s4,s5:s6,...];

PROCEDURES:      PROCEDURE p1; S;
                 SCALAR PROCEDURE sp1 (VALUE SCALAR vs1,vs2;
                       REFERENCE ROT rr1;
                       SCALAR ARRAY as1[2:3]);S;

OPERATIONS:
scalar s:     s+s,s-s,s*s,s/s,s↑s,|v|,|r|,|s|,v.v,s MAX s, s MIN s,s MOD s, s DIV s, INT(s)
vector v:     VECTOR(s,s,s),s*v,v*s,v/s,v+v,v-v,v*v,r*v,t*v,f*v,v WRT f,
                    UNIT(v),POS(f),AXIS(r)
rot    r:     ROT(v,s),r*r,ORIENT(f)
frame f:      FRAME(r,v),f+v,f-v,f*t,CONSTRUCT(v,v,v)
trans  t:     TRANS(r,v),f→f,t*t,INV(t)
boolean b:    ¬b, NOT b, b∧b, b AND b, b∨b, b OR b, b≡b, b EQV b, b⊗b, b XOR b,
                    s<s,s≤s,s=s,s≠s,s>s,s≥s
dimension d:  d*d,d/d,INV(d)

AL             s:    π,PI,BHAND,YHAND
CONSTANTS      v:    XHAT,YHAT,ZHAT,NILVECT
AND            r:    NILROT
VARIABLES      f:    BPARK,YPARK,STATION,BPARK,YPARK,⊗(valid only in MOVE)
               t:    NILTRANS
               b:    TRUE, FALSE
      units:         CM,INCH,INCHES,OUNCES,OZ,GM,LBS,SEC,
                     SECONDS,DEG,DEGREES,RADIANS
      dimensions:    DISTANCE,TIME,FORCE,ANGLE,TORQUE,ANGULAR_VELOCITY,VELOCITY,
                     DIMENSIONLESS

FUNCTIONS:
scalar       SQRT(s),SIN(s),COS(s),TAN(s),ASIN(s),ACOS(s),ATAN2(s,s),LOG(s),
             EXP(s),INSCALAR
boolean      QUERY(print list)

STATEMENTS:
comment      COMMENT <any text without semicolon);
             { <any text> }

control      FOR s← (scalar) STEP <scalar> UNTIL<scalar> DO (statement);
             IF (condition) THEN <statement> ELSE <statement);
             IF <condition> THEN <statement>;
             WHILE <condition> DO (statement);
             DO <statement> UNTIL (condition);
             CASE <scalar> OF BEGIN S;S;... S END;
             CASE <scalar> OF BEGIN [i1] S; [i2] S; ...ELSE j←0;[i3][i4] S END;

affix        AFFIX f1 TO f2 AT t1 RIGIDLY;
             AFFIX f3 TO f4 BY t2 NONRIGIDLY;
             AFFIX f3 TO f4 BY t2 AT t1 NONRIGIDLY;

unfix        UNFIX f5 FROM f6;
```

condition	ON FORCE(<vector>)<rel><force scalar> DO <statement>;	
monitor	ON TORQUE <rel> <torque scalar> ABOUT <vector> DO <statement>;	
statement	ON FORCE <rel> (force scalar> ALONG <axis vect> OF f1 DO ..;	
and	ON TORQUE <rel> <torque scalar> ABOUT <axis vect> OF f1 IN HAND DO ..;	
clauses	ON DURATION ≥ (time scalar> DO <statement>;	
	<label>: DEFER ON <event> DO <statement>;	
	<rel> is ≥ or <	
with	FORCE, TORQUE, DURATION similar to condition monitor	
clauses	WITH FORCE-FRAME = <frame> IN (co-ord sys>	
	WITH SPEED-FACTOR = <scalar>	
	WITH APPROACH = <distance scalar> or <distance vector> or <frame>	
	or DEPROACH (f1)	
	WITH DEPARTURE = ...	
enable	ENABLE <label>	
disable	DISABLE <label>	
deproach	DEPROACH(<frame>) ← <scalar> or <vector> or <trans> or <frame>;	
motion	MOVE f 1 TO <fval>;	
	MOVE f1 TO <frame> VIA <frame>,<frame>,<frame>;	
	MOVE f1 TO <fval>	
	VIA <frame) WHERE DURATION =<time scalar>,	
	VELOCITY = <velocity vector>,	
	<more clauses>;	
	MOVE f1 TO <fval> <more clauses>;	
	OPEN <hand> TO <distance scalar);	
	CLOSE <hand> TO <distance scalar>;	
	CENTER (arm);	
print	PRINT(<e>,<e>,...,<e>)	e is an expression, variable or string constant
abort	ABORT(<e>,<e>,...,<e>)	similar to print
prompt	PROMPT(<e>,<e>,...,<e>)	similar to print
pause	PAUSE <time scalar);	
signal	SIGNAL e1;	
wait	WAIT e1;	
assignment	<var> ← <expression>	
plan assign	<var> ←← <expression>	
require	REQUIRE SOURCE-FILE "DSK:FILE.EXT";	
	REQUIRE COMPILER_SWITCHES "LSK";	
	REQUIRE ERROR-MODES "LAMF";	
	REQUIRE MESSAGE "<message>";	
macro	DEFINE <macro-name> = ⊂<macro_body>⊃;	
	DEFINE <macro_name>(m1,m2,...)=⊂ <macro-body> ⊃;	
note	NOTE("<message>");	
	NOTE1 ("<message>");	
	NOTE2("<message>");	
dump	DUMP <variable-list>;	
return	RETURN	
	RETURN(<expression>)	

MACROS:	CRLF	carriage return and line food
	DIRECTLY	WITH APPROACH=NILDEPROACH WITH DEPARTURE=NILDEPROACH
	CAUTIOUS	SPEED-FACTOR ← 3.0
	SLOW	SPEED-FACTOR ← 2.0
	CAUTIOUSLY	WITH SPEED-FACTOR = 3.0
	SLOWLY	WITH SPEED-FACTOR = 2.0
	PRECISELY	WITH NULLING .
	APPROXIMATELY	WITH NO,NULLING

Appendix IV. POINTY COMMAND SUMMARY

```
DECLARATIONS:     SCALAR ts1, ts2;
                  VECTOR dv1,dv2;
                  ROT r1,r2;
                  FRAME f1,f2;
                  TRANS t1,t2;

OPERATIONS:
scalar  s:   s+s,s-s,s*s,s/s,|v|,|r|,|s|,v.v
             SQRT(s),SIN(s),COS(s),ASIN(s),ACOS(s),ATAN2(s,s),LOG(s),EXP(s)
             s MAX s,s MIN s,s MOD s,s DIV s,INT(s)
vector  v:   VECTOR(s,s,s),(s,s,s),s*v,v*s,v+v,v-v,r*v,t*v,f*v,v WRT f,
                  AXIS (v),UNIT(v),POS(f),v REL f
rot     r:   ROT(v,s),(v,s),r*r,ORIENT(f)
frame   f:   FRAME(r,v),(r,v),f+v,f-v,f*t,f1 REL f2, CONSTRUCT(f,f,f),CONSTRUCT(v,v,v)
trans   t:   TRANS(r,v),(r,v), (f→f), t*t

POINTY       s:   BHAND,YHAND
CONSTANTS    v:   XHAT,YHAT,ZHAT,NILVECT
AND          r:   NILROTN
VARIABLES    f:   BARM,YARM,BGRASP,BPARK,YPARK,STATION,BPARK,YPARK,
                  @(valid only in MOVE),FIDUCIAL, POINTER
             t:   NILTRANS


STATEMENTS:
comment      COMMENT <any text without semicolon>;
             { <any text> }

deletion     DELETE s1,s2,v1,v2,...;
             DELETE;
             QDELETE ALL;

function     FUNCTION ff1 = <expression);
             <type> FUNCTION ff2(<type>vt1,vt2; <type> vt3,vt4...)= <expression>

macro        DEFINE ml =⊂ (macro-body) ⊃;
             DEFINE m1 (mm1,mm2,...,mmn)=⊂<macro_body>⊃;

affix        AFFIX f1 TO f2 ;
             AFFIX f3 TO f4 NONRIGIDLY;
             AFFIX f3 TO f4 *;
             AFFIX f3 TO f4 AT t1 RIGIDLY;
             AFFIX f3 TO f4 AT t1 *;

unfix        UNFIX f5 FROM f6;
             UNFIX f5;

copy         COPY f1 INTO f2;
             COPY SUBTREE(f1) INTO f2;

motion       MOVE f1 TO (frame> + <vector exp>;
             MOVE f1 TO <f2>,<f3>,<f4>,...
             MOVE f1 BY <vector exp>;
             MOVEX fl BY <scalar>;
             OPEN <hand> TO <scalar);
             CLOSE <hand> TO <scalar);
```

```
                OPEN <hand> BY <scalar>;
                CLOSE <hand> BY <scalar>;
                DRIVE BJT(<jt no>) TO <scalar>;
                DRIVE YJT(<jt no>) BY <scalar>;
                CENTER <arm>;
                PARK; PARK BARM;

assignment <var>← <expression>

construct       FCONSTRUCT f

input/output    READ;        READ <file>;          QREAD <file>;
(file)          WRITE;       WRITE INTO. <file>;   WRITE <id>;
                WRITE ALL INTO <file>;             WRITE <id> INTO <file>;
                CLOSE;       CLOSE <file>;         CLOSE-FILES;
(terminal)      PRINT <exp>;
                SPRINT "<anything>";
                PROMPT;

edit            RENAME <var>;
rename          EDIT <var>;

display         DISPLAY SCALAR;
                REDISPLAY;
                NODISPLAY;

macro           DEFINE <macro-name> ≡⊂<macro_body>⊃;
                DEFINE <macro_name>(m1,m2,...)≡⊂ <macro-body> ⊃;
```

Appendix V. AL EXECUTION SUMMARY

```
        DEVICE      USER RESPONSE   AL RESPONSES

 1.     Terminal    Create file FOO.AL

 2.                 COMPILE FOO.AL

 2a.                                Swapping to SYS: AL. DMP
                                    AL: FOO 1 2 3 . . .

 2b.                                ALC

 2c.                                PALX 246

 2d.                                ALSOAP

 3.                 DO AL[AL,HE]

                                    f=
                    FOO
                                    ELF Assigned
                                    CORE SIZE=28K
                                    VERSION USING VT05
                                    GET SAV FILE -AL[AL,HE]
                                    OVERLAY BIN FILE -FOO
                                    AN EXTENDED COMMAND -VT05
                                    START AT (1000) (D FOR DDT) -D
                                    DDT STARTED AT 130000

 4.     VT05                        *
                    START <alt><alt> G
             or     <alt>G
                                        AL RUNTIME SYSTEM
                                        <any output from your program>
                                    ALL DONE NOW. SEE YOU AROUND!
                                    NO ACTIVE PROCESSES LEFT. YOU'RE IN DDT.
                                    *

 5.                 <alt>G
                    (for re-execution}
```

Appendix VI. AL examples

Here are several sample AL programs. A brief description is given for each of the programs given.

ENGINE ASSEMBLY

This program causes the arm to pick up a crankshaft assembly and lower it into t ho body of the engine. It then picks up the top of the engine and places it over the crankshaft, thereby completing the assembly.

```
BEGIN "assemble engine"
{ program to place the crankshaft assembly and the engine top on
       the engine body)

FRAME engine_top,engine_top_final,crankshaft,crankshaft_final;
FRAME bgrasp;

engine-top ← FRAME (ROT(ZHAT,90.000*DEGREES),VECTOR(51.0,34.2,3.13)*INCHES);
engine-top-final ← FRAME (ROT(ZHAT,45.000*DEGREES),VECTOR(57.3,49.3,10.2)*INCHES);
crankshaft-final ← FRAME (ROT(ZHAT,45.000*DEGREES),VECTOR(57.3,49.2,8.48)*INCHES);
crankshaft ← FRAME (ROT(ZHAT,90.000*DEGREES),VECTOR(51.3,40.3,4.09)*INCHES);

AFFIX bgrasp TO barm AT TRANS(ROT(xhat,180*degrees),nilvect*inches) RIGIDLY;

PRINT("ASSEMBLING ENGINE");

MOVE barm TO bpark WITH DURATION=3*seconds;
OPEN bhand TO 3.0*inches;       { initialize }

MOVE bgrasp TO crankshaft;
CENTER barm;                          { grab the crankshaft }

crankshaft ← bgrasp;

AFFIX crankshaft TO bgrasp RIGIDLY;

MOVE crankshaft TO crankshaft-final +3*zhat*inches
       WITH DURATION=2*seconds;       { take crankshaft above engine }

MOVE crankshaft TO crankshaft-final -0.3*zhat*inches
       WITH WOBBLE = 0.1*DEGREES
       WITH DURATION =5*seconds;      { insert piston }

UNFIX crankshaft FROM barm;
OPEN bhand TO 3.7*inches;             { release crankshaft }

MOVE bgrasp TO engine-top slowly;

CENTER barm;
engine_top←bgrasp;

AFFIX engine_top TO barm RIGIDLY;
MOVE engine-top TO engine_top_final+1.8*zhat*inches;
MOVE engine-top TO engine_top_final+1.0*zhat*inches;
                                      { by trial and error it was found
                                      that doing this reduced oscillation
```

```
                                        of crankshaft assembly }
MOVE engine-top TO engine-top-final -0.3*zhat*inches
     WITH FORCE-FRAME = STATION IN FIXED
     WITH WOBBLE = 0.1 *DEGREES
     WITH DURATION =5*seconds
     ON FORCE(zhat)≥80*ounces DO STOP engine-top;

UNFIX engine-top FROM barm;
OPEN bhand TO 3.8*inches;

MOVE bgrasp TO bgrasp + VECTOR(-4,-4,0)*INCHES;
                              { can't move out straight because elbow joint
                                (joint 5) will be at limit, so we move the
                                 hand sideways }

MOVE barm TO bpark;             { all done now }

END "assemble engine"
```

SHIFTING CASTINGS

This program causes the arm to shift a row of three castings back and forth between two positions.

```
BEGIN "casting shifter"
  {program to shift a row of three castings back and forth betwoon two positions}
  FRAME casting, casting-grasp, pick-up, set-down, line_1, line,2;
  DISTANCE VECTOR dpick, dset;
  SCALAR i,j,k;

  DEFINE TIL = ⊂STEP 1 UNTIL⊃;

  line_1 ← FRAME(ROT(zhat,90 * degrees),VECTOR(28,30,0) * inches);
  line_2 ← FRAME(nilrotn,VECTOR(32,24,0) * inches);

  AFFIX casting-grasp TO casting             (describe casting)
       AT TRANS(ROT(xhat,180 * degrees),VECTOR(1.2,1.5,1.87) * inches) RIGIDLY;

  MOVE barm TO bpark WITH DURATION = 4 * seconds;    (initialize arm)
  OPEN bhand TO 3.5 * inches;

  FOR k ← 1 TIL 2 DO                           (do it all twice}
    BEGIN "outer loop"
      pick-up ← line-l;
      sot-down ← line_2 - 0.8 * zhat * inches;          {initialize casting position}
      dpick ← -4 * yhat * inches;
      dset ←  ●  4 * xhat * inches;

      FOR j ← 1 TIL 2 DO         (move the castings there and back again}
        BEGIN "inner loop"

          casting ← pick-up;
          MOVE barm TO casting-grasp;              {go get first one)

          FOR i ← 1 TIL 3 DO                       {move three castings}
            BEGIN
            CENTER barm;
            casting-grasp ← barm;              (grab one}
            AFFIX casting TO barm RIGIDLY;
            MOVE casting TO set-down + 2 * zhat * inches     {shift it over}
                  WITH APPROACH = 2 * inches;
            MOVE casting TO set-down DIRECTLY                  {& place it on the table)
                  ON FORCE(zhat) ≥ 100 * oz DO STOP;
            OPEN bhand TO 3.5 * inches;                        (release it)
            UNFIX casting FROM barm;
            pick-up ← pick-up + dpick;                   (update where next one is}
            sot-down ← set-down + dset;                  {& where next goes}
            casting ← pick-up;
            IF i < 3 THEN MOVE barm TO casting-grasp          (go get next one}
                              WITH DEPARTURE = -3 * inches;
            END;

         pick-up ← line-2;            {got ready to move them back)
         set-down ← line-1 - 0.8 * zhat * inches;
         dpick ← +4 * xhat * inches;
         dset ← -4 * yhat * inches;
       END "inner loop";
    END "outer loop";

    barm ←← line_1 * FRAME(ROT(xhat,180 * degrees),VECTOR(-8,1,2) * inches);
    MOVE barm TO bpark WITH DEPARTURE = -4;            (when done put the arm away)

  END;
```

CASTING INSPECTOR

```
. BEGIN "casting inspector"
    (The arm is moved to a pick up point where it grabs a casting.
          Depending on the weight of the casting it is either rejected or accepted.
          Rejected castings are dropped in a garbage bin, while accepted ones
          are lined up in a row.  The program terminates after finding three
          good castings.)
    FRAME pick-up, set-down, garbage, casting, casting-grasp;
    DISTANCE VECTOR dset;
    SCALAR good-castings, heavy;

    sot-down ← FRAME(nilrotn,VECTOR(15,40,-0.8) * inches);        {initial locations}
    pick-up ← FRAME(ROT(zhat,90 * degrees),VECTOR(4,44,0) * inches);
-   garbage ← FRAME(ROT(zhat,90 * degrees),VECTOR(18,45,7) * inches);
    dset ← -4 * xhat * inches;

    AFFIX casting-grasp TO casting                   {describe casting)
          AT TRANS(ROT(xhat,180 * degrees),VECTOR(1.2,1.5,1.87) * inches) RIGIDLY;

    MOVE barm TO bpark WITH DURATION = 4 * seconds;       (initialize arm position}
    OPEN  bhand TO 3.5 * inches;

    good-castings ← 0;

    DO                  {loop to find 3 good castings]
      BEGIN

        casting ← pick-up;
        MOVE barm TO casting-grasp;                (go get a casting)
        CENTER barm;
        casting-grasp ← barm;
        AFFIX casting TO barm RIGIDLY;
      heavy ← false;                               {see if it weighs enough)
      MOVE casting TO ⊗+2*zhat* inches DIRECTLY
         ON FORCE(-zhat)≥ 85 * oz DO heavy ← true;

      IF heavy THEN
         BEGIN "good casting"
         MOVE casting TO set-down • 2*zhat* inches DIRECTLY;
         MOVE casting TO set-down DIRECTLY
              ON FORCE(zhat)≥ 90 *oz DO STOP;. {place it on table with others}
         OPEN bhand TO 3.5 * inches;
         UNFIX casting FROM barm;
         good-castings ← good-castings + 1;          (update # good ones}
         set-down ← set-down +dset;                  {& where next goes)
         MOVE barm TO ⊗+3 * zhat * inches;
         END "good casting"
        ELSE
         BEGIN "bad casting"
         MOVE casting TO garbage DIRECTLY;
         OPEN bhand TO 3.5 * inches;                 (trash bad one}
         PRINT("defective casting!",crlf);
         UNFIX casting FROM barm;
         END "bad casting";

    END
   UNTIL good-castings ≥ 3;              (repeat until we find 3 good castings}

   barm ←←FRAME(ROT(xhat,180*degrees),VECTOR(12,41,2)* inches);
   MOVE barm TO bpark DIRECTLY;                      (put arm away when done}

  END;
```

VISION INTERACTION

This program utilizes vision through an ALAID interface. The AL program runs on the PDP- 11, waits for the vision program on the PDP-10 to take a picture of a human hand and compute the coordinates of the center, On obtaining the center of the hand, the AL program causes the blue arm to pick up a tool from the workplace and move it over to the human hand. Both the AL program and the SAIL program which incorporates the vision routines are listed below.

```
BEGIN "tool"
   COMMENT This program does SIGNALs events to and WAITs for events from its
      counterpart residing on the pdp 10 through an ALAID interface ;


   EVENT al, sail;
                   { AL will SIGNAL sail, and WAIT al;
                     SAIL will SIGNAL al, and WAIT sail)
   DISTANCE SCALAR handx,handy,handz;
                   {x,y,z coordinates in inches of the center of
                     person's ieft palm facing the camera.
                     The plane of the palm should be about flush
                     with the edge of the table }
   handx← -1 .73*INCHES;
   handy+ 42.9*INCHES;
   handz←11.8*INCHES;{ about mean position for human hand position }
   MOVE barm TO bpark WITH DURATION=3*seconds;{ park the arm }
   OPEN bhand TO 2.5*inches;

   WHILE truo DO
       BEGIN "infinite loop"
       DISTANCE VECTOR hand-pos;
       FRAME tool,tool_tip,tool_store,person;

       SIGNAL sail; { tell vision program to look for victim's hand }

       PRINT("WAITING FOR VISION TO GIVE ME LOCATION OF HUMAN HAND");

       WAIT al;        { waiting for SAIL to signal to AL
                       after verification vision has updated values of
                       handx,handy, and handz}

       hand-pos+VECTOR(handx,handy,handz);
                       { update vector giving location of human hand }
       tool-store ← FRAME (ROT(YHAT,180.0*DEG),VECTOR(52.996,45.042,2.835)*INCHES);
       tool ← tool-store;
       AFFIX tool-tip TO tool AT TRANS(NILROTN,ZHAT*2*INCHES);
       person ← FRAME (ROT(ZHAT,151.93*DEG)*ROT(YHAT,125.22*DEG)*ROT(ZHAT,-141.13*DEG),
              hand-pos);

       { now let us do the actual motions }
              barm←←bpark; bhand←←2.5*inches;     { so world modeller will not complain }
              tool←tool_store;
              MOVE barm TO tool SLOWLY;
              CLOSE bhand TO 0.0*INCHES APPROXIMATELY;
              AFFIX tool TO barm RIGIDLY;
              MOVE tool-tip TO person
                     WITH APPROACH=NILDEPROACH APPROXIMATELY SLOWLY;
              PAUSE 2*seconds;
              OPEN bhand TO 2.5*inches APPROXIMATELY;
              UNFIX tool FROM barm;
              PAUSE 1 *seconds;
              MOVE barm TO bpark;
                     { move arm out of view of camera }
  END "infinite loop";
END;
```

The following is an extract of a SAIL program which shows the parts relevant to the ALAID communication link. This program runs on the POP-1 0 while the previous program is running concurrently on the POP-1 1.

```
BEGIN "vision program"

PRINT("CALLING ALAID ");
TREATREQUEST("SYMBOLS TOOL.ALS[DEM,HE]",1);
       COMMENT treatrequest is the ALAID call.
              This call reads in the symbol table for the AL program ;

       COMMENT At this point assign nominal values of the human hand position
              to variables y0,z0;
  WHILE true DO
       BEGIN "more pictures"
       REAL newy,newz,dy,dz,theta;
       PRINT("Waiting for 11 to Signal me -");
```

```
        TREATREQUEST("WAIT (NAME SAIL)", 1);
              COMMENT Here we wait for the 11 to tell us it is ready;

        WHILE NOT LOCATE(theta,dy,dz)
           DO PRINT("Unsuccessful...Trying again");
              COMMENT LOCATE takes the picture and computes the
                    angular offset theta and the y0 and z0 offsets
                    dy, dz from the nominal position y0,z0 which has been
                    defined earlier. The value of the function
                    is true if the picture can be matched, otherwise
                    it is false ;

        COMMENT At this point vision has successfully found dy,dz, theta;

        newy←dy+y0; newz←dz+z0;
        PRINT("Sending AL new Y,Z: ",newy," ",newz,crlf);

        COMMENT Now let us set the values of handy and handz;
        TREATREQUEST("SETVAL (NAME HANDY) (SCALAR "&cvf(newy)&")",1);
        TREATREQUEST("SETVAL (NAME HANDZ) (SCALAR "&cvf(newz)&")",1);

        COMMENT Now tell the 11 we are ready ;
        TREATREQUEST("SIGNAL (NAME AL)",1 );
        END "more pictures";

END;
```

Appendix VII. POINTY examples

LOCATEZ_UP

LOCATE_ZUP is used to determine the x and y coordinates of the axis of an upright cylinder. The macro tells the user to move the arm to the approximate location of the object, and then it does a center, reads the hand position, opens the hand, rotates it 90 degrees, closes it again and takes a second reading, and then produces a frame with stat ion orientation. A similar macro which rotates the wrist 60 degrees is used fo hexagonal cylinders.

```
DEFINE LOCATE_ZUP(ACTUAL_POS)=⊂
       SPRINT " MOVE ARM TO APPROX POSITION OF ACTUAL,POS ";
       PROMPT;              { lets user prompt when he is ready }
       CENTER BARM;         {use sensing to get position }
       OPEN BHAND TO BHAND_MAX;
                    {BHAND_MAX has boon defined elsewhere as 3.8 inches }
       MOVE BGRASP TO FRAME(ORIENT(↑BARM)*ROT(ZHAT,90),POS(BGRASP));
                    { so now we move the arm so that the wrist is rotated
                    90 degrees but the hand points vertically downwards }
       CENTER BARM;
       ACTUAL_POS←FRAME(NILROT,POS(BGRASP));
                    { and we determine the final position of the object
                    but give it station orientation }
       OPEN BHAND TO BHAND_MAX;
       MOVEZ BGRASP BY 3*inches;   { open hand and get the arm out of the way }
       ⊃;
```

MOVE-AND-READ

This macro is used to ask the user to move the arm to a certain location. The position is then recorded.

```
DEFINE MOVE_AND_READ(POSITION) =
       ⊂
       { simple macro that asks user to move arm to a position and records it }
       SPRINT "MOVE ARM TO POSITION";
       PROMPT;
     POSITION* BARM;
     · ⊃;
```

MOVEMACR03

This macro is used to define three positions and a new macro that will make the arm go through these positions. It illustrates the use of the MOVE-AND-READ macro, and may be used to teach motion through a series of three frames that will avoid obstacles in its path.

```
DEFINE MOVEMACR03 (MACNAME,P1,P2,P 3) .
        ⊂
        MOVE_AND_READ(P1);
        MOVE_AND_READ(P2);
        MOVE_AND_READ(P3);
        DEFINE MACNAME =
                ⊂
                MOVEBARM TO P1,P2,P3;
                ⊃;
        ⊃;
```

Appendix VIII. REFERENCES

Binford,T.O., Grossman,D.D., Liu,C.R., Bolles,R.C., Finkel,R.A., Mujtaba,M.S., Roderick,M.D., Shimano,B.E., Taylor,R.H., Goldman,R.H., Jarvis,J.P., Scheinman,V.D., Gafford,T.A.; *EXPLORATORY STUDY OF COMPUTER INTEGRATED ASSEMBLY SYSTEMS;* Progress report 3 covering period from December 1, 1975 to July 31, 1976. Stanford Artificial Intelligence Laboratory MEMO AIM-285, Computer Science Department Report STAN-CS-76-568 Standford University. August 1976.

Binford,T.O., Liu,C.R., Gini,G., Gini,M., Glaser,I., Ishida,T., Mujtaba,M.S., Nakano,E., Nabavi,H., Panofsky,E., Shimano,B.E., Goldman,R., Scheinman,V.D., Schmelling,D., Gafford,T.; *EXPLORATORY STUDY OF COMPUTER INTEGRATED ASSEMBLY SYSTEMS;* Progress report 4 covering period from August 1, 1976 to March 31, 1977. Stanford Artificial Intelligence Laboratory MEMO AIM-285.4, Computer Science Department Report STAN-CS-76-568 Standford University. June 1977.

Finkel,R., Taylor,R., Bolles,R., Paul,R., Feldman,J.; *AL, A PROGRAMMING SYSTEM FOR AUTOMATION*; Stanford Artificial Intelligence Laboratory MEMO AiM-243 Computer Science Department Report STAN-(X-74-456 Standford University. November 1974.

Finkel,R.A.; *CONSTRUCTING AND DEBUGGING MANIPULATOR PROGRAMS;* Stanford Artificial Intelligence Labratory MEMO AIM-284, Stanford Computer Science Department Report STAN-CS-76-567 Stanford University. August 1976.

Grossman,D.D., Taylor,R.H.; *INTERACTIVE GENERATION OF OBJECT MODELS WITH A MANIPULATOR*; Stanford Artificial Intelligence Laboratory Memo AIM-274, Stanford Computer Science Report STAN-CS-75-536 Stanford University. December 1975.

Paul,L.; *MODELLING, TRAJECTORY CALCULATION AND SERVOING OF A COMPUTER CONTROLLED ARM;* PhD. Dissertation, Stanford Artificial Intelligence Labrotary Memo AIM-1 77, Computer Science Department Report STAN-CS-72-311 Stanford University. March 1973.

Shimano,B.E.; *THE KINEMATIC DESIGN AND FORCE CONTROL OF COMPUTER CONTROLLED MANIPULATORS*; Stanford Artificial Intelligence Laboratory MEMO AIM-313, Stanford Computer Science Department Report STAN-CS-78-660 Stanford University. March 1978.

Taylor,R.H.; *A SYNTHESIS OF MANIPULATOR CONTROL PROGRAMS FROM TASK-LEVEL SPECIFICATIONS;* Stanford Artificial Intelligence Laboratory MEMO AIM-282, Computer Science Department Report STAN-CS-76-560 Stanford University, July 1976,

Appendix IX. ACKNOWLEDGEMENTS

We wish to acknowledge the following contributors who have brought the AL system to its current state of development: Tom Binford, Bob Bolles, Ray Finkel, Jerry Feldman, Tom Gafford, Maria Gini, Pina Gini, Dave Grossman, Norman Haas, Tatsuzo ishida, Pitts Jarvis, Bill Laaser, Hamid Nabavi, Lou Paul, Ted Panofsky, Gene Salamin, Ken Salisbury, Mike Roderick, Vic Scheinman, Bruce Shimano, Russ Taylor.

This work was supported by the National Science Foundation through grants NSF-APR-74-01390-A04 and NSF-DAR-78-1 5914.