CAOS
An Approach to Robot Control

Nils Thune and Bir Bhanu

Computer Science Department
University of Utah
Salt Lake City, Utah 84112, USA

UUCS-87-007

31 March, 1987

Abstract

Control systems which enable robots to behave intelligently is a major issue in todays process of automating factories. This thesis presents a hierarchical robot control system, a programming language for goal achievement, termed CAOS for Control using Action Oriented Schemata, with ideas taken from the neurosciences. The system uses action oriented schemata (neuroschemata) as the basic building blocks in a hierarchical control structure. Serial versions in C and LISP are presented with examples showing how CAOS achieves goals. Moreover, a partial implementation of a parallel version of the system is discussed.

This work was supported in part by NSF Grants DCR-8506393, DMC-8502115, ECS-8307483 and MCS-8221750.

Table of Contents

1. From Neuron to Neuroschema 4
1.1 Introduction 4
1.2 The Neuron 4
1.3 The Schema 6
1.4 The Neuroschema 7
1.4.1 The Activation Section 7
1.4.2 The Event Section 8
1.4.3 The Learning Section 10
1.5 Conclusion 10
2. CAOS: A Hierarchical Control System 12
2.1 Introduction 12
2.2 Action Oriented Control 13
2.3 The Global Knowledge Base 13
2.4 The Global Data Base 15
3. Exploiting Parallelism in CAOS 17
3.1 Introduction 17
3.2 Parallelism in the Human Brain 18
3.3 Parallelism in the Control System 18
3.4 Parallelism in Programs 19
3.5 Processor Utilization 19
4. CAOS Versus Expert Systems 20
5. An Overview of CAOS 22
5.1 Introduction 22
5.2 Syntax and Meaning of CAOS clauses 28
5.2.1 AND clauses 28
5.2.2 OR clauses 28
5.2.3 Function clauses 30
5.2.4 Expected pre- and post-inputs clauses 31
5.2.5 Output clauses 31
5.3 CAOS Commands 32
5.3.1 Achieve 32
5.3.2 Consult 32
5.3.3 Display 32
5.3.4 Erase 33
5.3.5 Help 33
5.3.6 Reconsult 33
5.3.7 Trace-on 33
5.3.8 Trace-off 33
5.4 Example: The Cube of a Number 34
5.5 Example: Graphics Demo 36
6. Conclusions and Future Work 39
7. Appendix A 40
8. Appendix B 41

List of Figures

Figure 1-1: The Basic Structure of a Neuron 5
Figure 1-2: The Schema Components 6
Figure 1-3: The Neuroschema Components 8
Figure 1-4: The Neuroschema Depicted as a Neuron 9
Figure 2-1: The CAOS System Structure 12
Figure 2-2: Hierarchy of Goals and Subgoals 14
Figure 2-3: Structure of Node and Leaf Objects/Nodes 15
Figure 2-4: Data Base Objects 16
Figure 4-1: The Structure of an Expert System 20
Figure 5-1: Goal: (put-on (a b)) 22
Figure 5-2: Goal: (get-space (a b)) 24
Figure 5-3: Goal: (find-space (a b)) 25
Figure 5-4: CAOS's Knowledge Base After The Consult Command 27
Figure 5-5: AND nodes 29
Figure 5-6: OR nodes 29
Figure 5-7: Leafs 30
Figure 5-8: Different ways of obtaining the goal, (cube (x)) 35

Preface

As computers become cheaper and more compact, and the availability of high quality sensors increases, it becomes attractive to create intelligent robots for use in automated environments for recognition, assembly, inspection, and manipulation of objects [16]. Due to safety hazards, repeatability of tasks, or economic constraints, it is an attractive notion to replace humans with robots in many of the tasks mentioned. The availability of hardware for intelligent robots creates a need for designing control programs which have the capability of intelligent goal seeking. The control needs to be concerned with goal achievement guided by diverse information from multiple sensors such as TV cameras, range finders, and tactile-, force-, and torque-sensors. Control becomes crucially important as the tasks, enabled by multisensors, becomes more complex and involved. Consequently, a control system needs to be flexible, adaptable, and able to learn from experience.

The processing involved in a control system used for robots in automated environments often needs to be done in real time, and it is therefore natural to bring parallel processing into the picture, enabling considerable speedup in execution time when compared to sequential processing on conventional processors. For example, low level image processing, involving large amounts of data, is often accomplished in real time using parallel processors. Furthermore, the control can also experience speedup by achieving independent subgoals in parallel.

Many existing robot control systems assume a very restricted operational environment [1, 4, 10] limiting the usefulness of the system to a small domain or to tasks which follow a particular pattern in a repetitive fashion. In many cases, for example spray-painting, this is quite adequate. For many other tasks in less structured environments the robots need to be more sophisticated.

Knowledge about the intelligent aspects of a control system can be drawn from the neurosciences where studies of the most intelligent system we know, the brain and nervous system, indicate some important and basic factors of our intelligence [1, 19]. These factors are also important for a robot control system:

- The brain is made of basic building blocks, called neurons.
- The brain is structured in a hierarchical manner.
- The brain operates in parallel.

The neurons process and produce information which is used to make intelligent decisions about tasks to be done. Even though there are many categories of neurons, such as motor neurons and sensory neurons,

almost all of them have the same general structure [19, 20]: multiple *dendrites* carry the input to the *cell body* where the information is processed, and a single *axon* carries the output to other neurons in the nervous system. All of the neurons, with their dendrites and axons, are organized into a complex network which is probably the key to our intelligence, since it provides the necessary links between parts of our brain [1, 2, 19, 20].

It is believed that the neurons, with their complex network of interconnections, are organized in a hierarchical fashion [1]. Commands are issued at the top, and are split into subgoals as they propagate down the hierarchy. In addition to the hierarchical organization, the brain makes extensive use of parallelism in carrying out its tasks [1, 2, 9, 11]. Many neurons operate in parallel, receiving input, processing it, and propagating the results to many other neurons. The brain is quite slow compared to digital computers, being able to carry out only about 100 serial time steps per second [9, 11]. The normal reaction time for a human being is approximately 0.5 to 1.0 seconds, and the tasks which the brain carries out during this time often requires a substantially higher number of computations than 100, leading to the conclusion that parallelism is essential for our ability to react as fast as we do.

In developing an intelligent control system for robots, it is desirable to include the three important aspects of the brain already discussed: basic building blocks, hierarchical organization, and parallel processing. With this in mind, an approach to robot control called *Control using Action Oriented Schemata* or *CAOS* is presented in this thesis. The action oriented schemata are termed *neuroschemata* because of their similarity to *neurons*, which are the basic building blocks of the brain, and *schemata* [3, 14, 24] which are a basic kind of representation. Each neuroschema is able to activate several other neuroschemata in parallel, and they are the basic building blocks of the control system. The neuroschemata are organized in a hierarchical manner for each goal the system can achieve. Hence, three of the main aspects of the brain have their analogs in CAOS: basic building blocks, hierarchical organization, and parallel processing.

The purpose of CAOS is to achieve high level goals, specified by a user, through planning and action. The goals which can be achieved depend upon the system's knowledge base, and are restricted by existing rules, facts, and procedures which the system can consult.

Two serial versions implemented in C and LISP respectively is presented. The C version [8] is the preliminary version and its knowledge is sufficient to locate and recognize simple polyhedral objects in range images. Due to implementation difficulties with the C version and the prospect of a LISP compiler

for the BBN Butterfly Parallel Processor, the next version was written in LISP [7].

Both the C and the LISP serial versions of CAOS was developed keeping in mind that they should be easily transportable to the BBN Butterfly.

1. From Neuron to Neuroschema

1.1 Introduction

To enable robots to interact intelligently with their environment, we need an artificial brain that can control the robot. Such a "brain" can be modeled after the human brain.

The human brain and nervous system controls the activities of the human body. It coordinates the various activities, receiving thousands of bits of information from multiple sensor organs and interneurons [12, 17, 19]. Computational responses to the environment, which the body exists in, originates from sensory experience enabled by various tactile, auditory, and visual receptors. Neurons, the fundamental units of the nervous system, encode and decode complex information through a network of interconnections between millions of other neurons. The interconnections and processing between neurons forms the intelligent system that permeates the whole body. In developing an intelligent control system it is natural to use ideas found in studying the brain. Hence, some kind of basic building block for the robot control system is sought.

1.2 The Neuron

Neurons exist in millions, arranged in regular patterns and grouped in functional divisions, in the brain. They are analog computing devices that can take on any number of inputs, and produce an output that can function as one input to hundreds of thousands of other neurons. Neurons are structured in a hierarchical manner using extensively their ability to process signals in parallel [17, 19]. They are richly interconnected, each receiving inputs from several other neurons or receptor cites throughout the body.

The basic structure of a neuron is shown in Figure 1-1. There exists several different types of neurons, but they all have the same general structure.

The *denrites* provides the informational input to the neuron. They branch out to other neurons and/or receptors and receive signals which they carry back to the *cell body* of the neuron. The dendrites can provide the neuron with both inhibitory and excitatory signals.

The *cell body* provides the "function" of the the neuron. It encapsules the nucleus and provides the necessary energy to keep the neuron alive. The cell body monitors input from its denrites and it produces an output routed through the axon. The function performed in the cell body can be as simple as summing up the different inputs at a particular moment and producing that as an output, or as complex as a non-

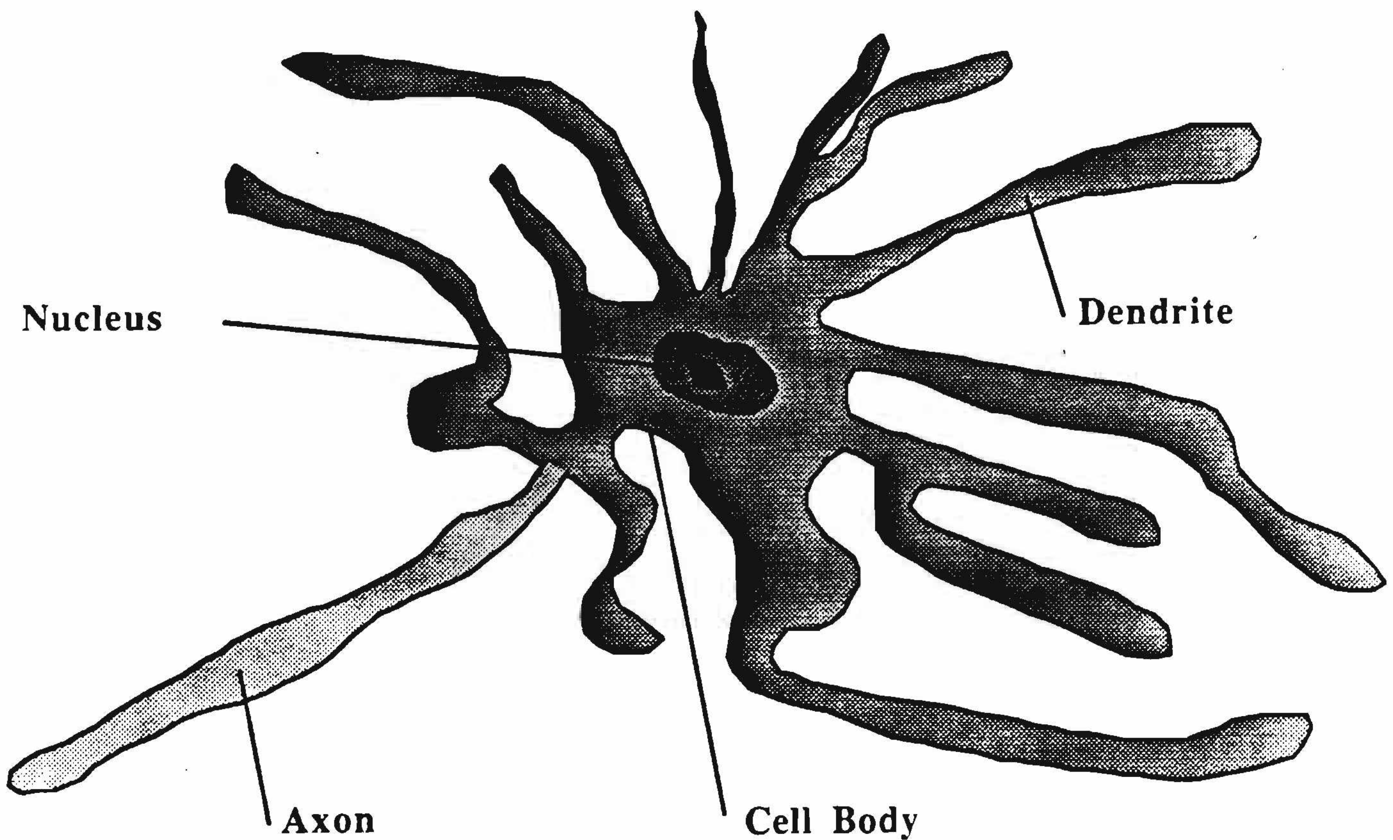

Figure 1-1: The Basic Structure of a Neuron

linear function that take into account signal levels and time differences for each input before any output is produced.

The *axon* is responsible for carrying the output of the neuron to other parts of the brain and nervous system. The dendrites usually receive their signals from the axons via the cell body (axo - axonal, dendro - axonal, and dendro - dendritic connections also exist). Depending on the type of neuron the axon can extend micrometers out from the cell body, inside the brain, to a meter in length when branching out to receptor sites on the human body.

Each neuron is functioning by itself. The neuron can be viewed as constantly being in a particular goal state. It receives input from multiple receptors and/or interneurons, and the cell body processes this information and computes an output, a goal state.

1.3 The Schema

A schema can be viewed as an abstract data type with sensory processing, action, and possibly learning elements [3, 14, 24]. It is, *in theory*, similar to the neuron and its functionality.

A schema monitors certain aspects of the current situation and becomes active when the situation matches an expected state. It is both a process and a representation. The major components of the schema are shown in Figure 1-2. The actions and sensing performed by schemata include modifying and monitoring the internal state of the system as well as the environment. Schemata can serve as building blocks for both representations and programs. At the highest level of control, schemata are used for planning, and at the lowest level they provide servo control with sensory processing at all levels.

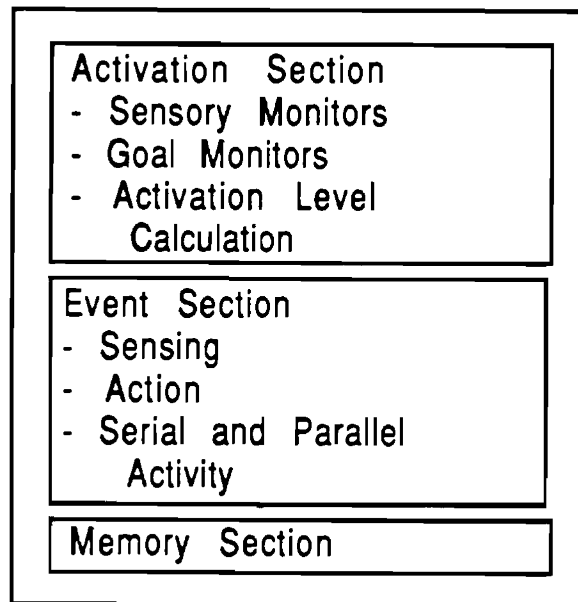

Figure 1-2: The Schema Components

The activation section is the control center of the schema. It activates the schema when a certain level is calculated and exceeds a predefined threshold for action. It monitors the current state of the schema and the surrounding environment to turn the schema on and off. The event section specifies the action and sensing to be performed when the schema is active. The memory section is intended to be an adaptation mechanism for the schema.

Clearly, the schema can not be used for implementation purposes with present software and hardware technologies. Each schema's activation section is supposed to constantly monitor the current situation, which implies that it needs to occupy a single processor alone. In a control system for a robot where one

would incorporate thousands of schemata to enable stable control, it is clear that schemata are not implementable. For practical purposes another type of basic building block is needed which does not rely on unlimited parallel processing capabilities.

1.4 The Neuroschema

Due to the lack of parallel processing being available to the extent needed to fully simulate all the neurons in the brain, a more rigid approach to control is needed. Some processing can be done in parallel on existing computers, but not enough compared to the amount needed for simulating the brain, and hence a different basic building block is expected for the control system if it is to be functionally implemented on todays existing parallel processors.

The neuroschema provides this basic building block. It is a strict processing element for controlling input and output of a specific function. The neuroschema can be viewed as corresponding to the cell body of the neuron, with the difference that it only becomes active in certain situations, contrary to the neuron which is active constantly.

The neuroschema consists of three sections: an *Activation Section*, an *Event Section*, and a *Learning Section*. Each neuroschema is a LISP *method*, simulating the general functions of a neuron by receiving input, processing it, and producing output based upon the function (goal) it controls. Figure 1-3 shows the major components of the neuroschema.

A neuron can take on any number of inputs and produces an output. The same is true for the neuroschema. In addition, *any type* of input and output can flow through it, making it flexible. The neuroschema is activated with a goal (a LISP object) and uses the information in this goal to determine how to process the input and output. An active neuroschema with its goal in the robot control system is the counter part to the neuron of the human nervous system. Figure 1-4 shows the neuroschema in a neuron-like fashion.

1.4.1 The Activation Section

Information flowing in the control system is provided by the user, multi sensors [22], and computations done internally by the neuroschemata. In contrast to neurons, which constantly monitors its dendrites (inputs), the neuroschema only becomes active when a particular goal state is requested. To achieve this particular goal state a neuroschema is activated with a goal object which contains information about how to obtain this goal. The *activation section* of a neuroschema becomes active at the moment the neuros-

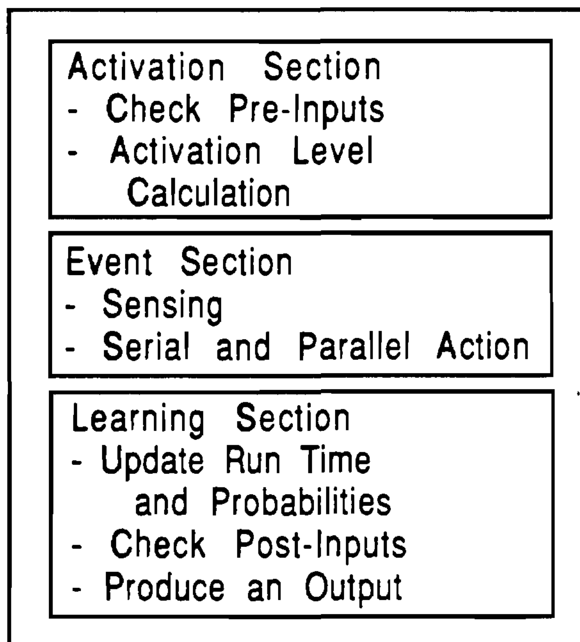

Figure 1-3: The Neuroschema Components

chema is activated. Along with the goal object the neuroschema is also activated with some particular input which we call *pre-input* (as opposed to post-input discussed later). Using the information found in the goal object the neuroschema checks the pre-input for acceptable range and type. Depending on this result the event section of the neuroschema is activated.

1.4.2 The Event Section

The *Event Section* of the neuroschema uses the information in the goal object to determine how the goal is to be achieved. The neuroschema, activated with a goal, obtains this goal by achieving its subgoals. The subgoals are obtained by either activating a new neuroschema for each of them, or if the subgoal simply is a program/function, it is executed. Subgoals of a goal can be directly compared to interneurons of the brain while programs/functions are the same as motor neurons in the nervous system controlling receptors and joints.

One particular goal can sometimes be obtained by achieving either one of several different, alternative subgoals (an OR-node). Using statistics and average run time from previous successes in obtaining the different goals, the goal with highest expectation of success is chosen. Success means that the (sub)goal has been obtained with satisfactory output and failure indicates that the (sub)goal was not achieved. This

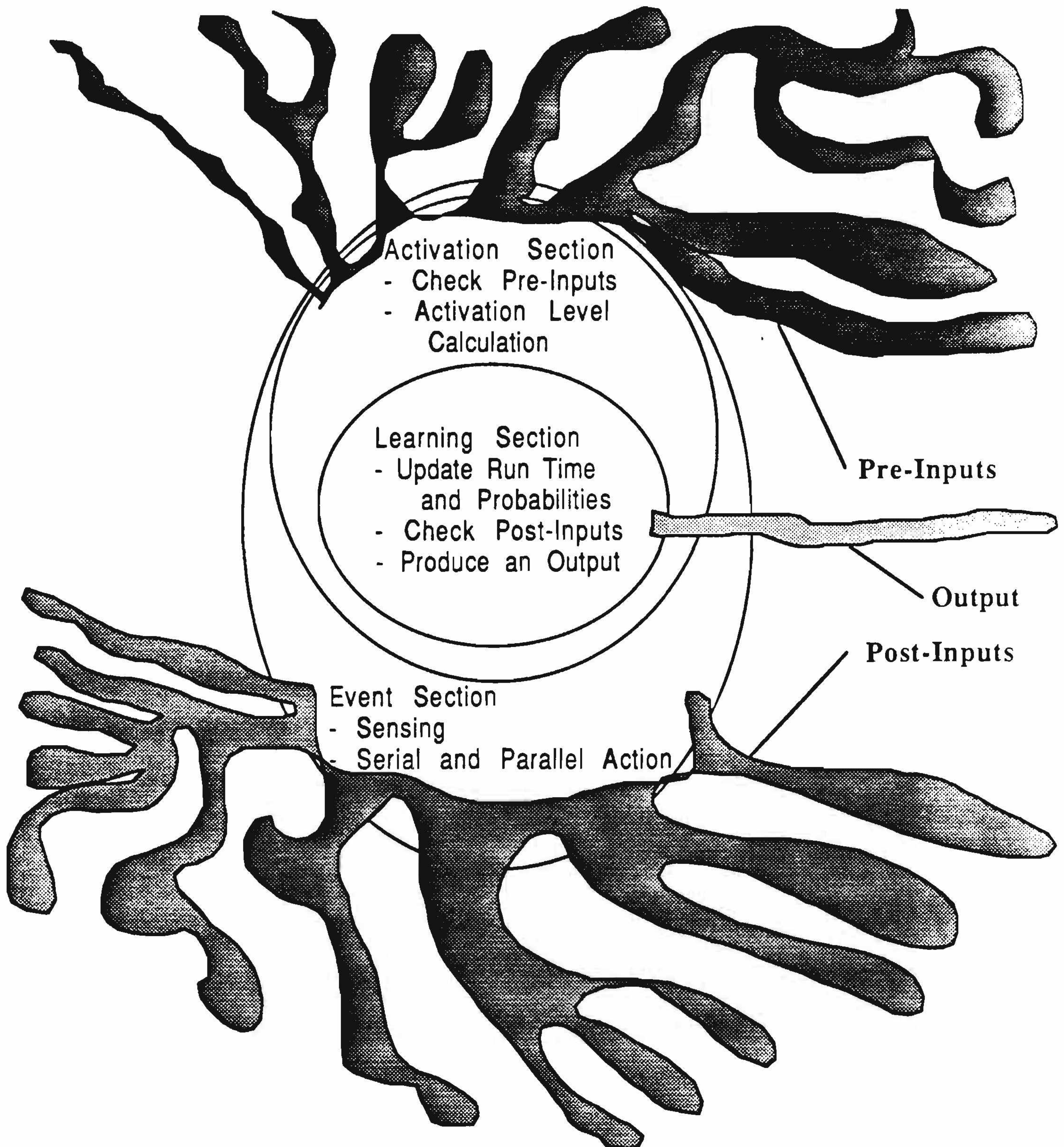

Figure 1-4: The Neuroschema Depicted as a Neuron

is comparable to the function of the cell body of a neuron which produces an output when necessary input satisfies it (the goal state is met).

If a (sub)goal has never been active before, the expectation of obtaining it is said to be 0.0 (NOTE: E(s) is defined over the range 0.0 - ∞ where 0.0 denotes the highest expectation for obtaining the goal). In all other cases we have:

$$E(s) = (\text{Average run time of goal})/P(s)$$

The average run time of a goal is given as the total sum of all run times for the goal, when the goal were achieved, divided with S (the number of successes for obtaining this goal). P(s) is given by S divided with N (the total number of trials for this goal).

If there is more than one alternative for obtaining a goal, the control system choses the subgoal with the highest expectation of success. If the expectation values are equal, the leftmost branch is chosen, or in a parallel version, both would be pursued when the expectation values are within a specified threshold.

1.4.3 The Learning Section

The third section in the neuroschema, the *Learning Section*, is activated when no information about how to obtain the main goal can be found in the knowledge base, indicated by the scheduler. It is also activated each time a neuroschema has obtained all its subgoals or has executed its procedure to update the average run time and probability of success. Finally, this section checks the post-inputs and produces the output of the goal based on these. Post-inputs are the output of the neuroschemata controlling the subgoals of a goal. Hence, post input and pre-inputs together is comparable to all the input received through the denrites of a neuron.

1.5 Conclusion

The structure of the neuroschema resembles the schema of Arbib, Iberall and Lyon, and Overton [3, 14, 24], but is very different in functionality. In contrast to their schemata, which have preconstructed plans for achieving a goal, the neuroschema is a control environment which can be activated with any plans for goal achievement from the knowledge base. Another difference is found in the approach to learning, which, in the case of schemata, is done by instantiating new schemata which better fit a new situation. When this system is learning, new information about how to achieve the goal is used to update the knowledge base. This new information takes the form of probability and average run time measures, and is used to achieve the goal the next time. Furthermore, neuroschemata is implementationally possible on current equipment, using parallel processing as available, while schemata are directly restricted in operation depending on processor availability.

Overall, the neuroschemata provide the functionality of neurons, incorporating parallelism as it is available, and is easily implementable with no requirements to the system it is implemented on.

2. CAOS: A Hierarchical Control System

2.1 Introduction

An intelligent control system for a robot needs to be able to achieve multisensory coordination. Moreover, for a control system to act intelligent it needs the power of perceiving, learning, knowing, and reasoning. Hence, to make a robot intelligent it must be capable of the following:

- It must be able to perceive. That is, it should be able to become aware of things through receptors (multisensors) and previous knowledge.
- It must be able to learn and know; It should have the capability to learn by receiving information from the environment and use this as knowledge later.
- Finally, it must be able to understand and reason. That is, it should understand the meaning and/or nature of a problem and be able to use logic to solve it.

The ability to process sensory data in a consistent way is important and to fully utilize the resulting information it must be integrated into the control of the robot.

A control system for a robot can be viewed as having two parts. First, the structure which controls the sensory and motor behavior of the robot and secondly the processing and representation of the sensory and control data.

CAOS is a framework for developing goals and representing knowledge and is able to coordinate such goals based on multisensor information and basic environmental knowledge. The overall system structure of CAOS is shown in Figure 2-1.

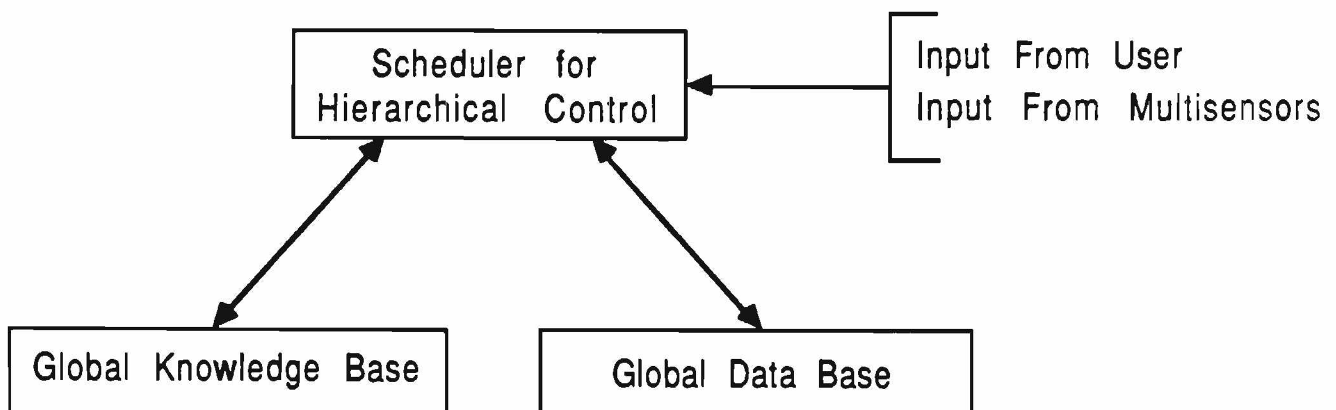

Figure 2-1: The CAOS System Structure

With CAOS, one is able to represent a dynamic world in which the robot interacts with the environment through planning and action based upon goals the robot is capable of performing. The approach used in CAOS provides control at all levels extending from specifying low level motion, position, velocity, and force commands on joints and end effectors to high level planning.

2.2 Action Oriented Control

As control programs grow exponentially in complexity with the number of sensors and branch points in the plan for achieving a goal, the choice of using hierarchical control with action oriented schemata (neuroschemata) enables the system to be partitioned into levels of limited complexity. Each level in the hierarchically organized tree have goals which decrease in complexity with decreasing level in the hierarchy. Each level in the tree produces "a jump in the intelligence" of the the system. Figure 2-2 shows an example of this.

Direct evidence of the hierarchical subdivision of control is found in the brain and is called the *Neuronal Hierarchy* [1]. At the bottom, neuronal computations are concerned with only a single muscle group. Moving one level up, the computation now involves several muscle groups. Several levels above this again the motion of an entire limb, and action between limbs, are computed. Finally at the highest level of the neuronal hierarchy, the entire human body is coordinated towards future goals.

CAOS is structured in a hierarchical way and is action oriented (the systems goals knows which subgoals to activate to achieve a specific goal state). It is based upon basic building blocks, neuroschemata, which incorporate mechanisms for planning, stable control, sensory processing, and representation of the world.

The CAOS shell controls the user interface and goal achievement. It enables the user to describe goals which the system should be able to achieve.

2.3 The Global Knowledge Base

The knowledge base is a part of the *world model* of the control system, and can be viewed as an analog to the *long term memory* [4] of the human brain. This is in contrast to the *short term memory*, which uses information found in long term memory to obtain a goal, and then disappears.

The knowledge base contains three *goal* types. They are LISP objects classified as *and-nodes*, *or-nodes*, and *program-leafs*. These three types represent goals that CAOS can achieve. They are

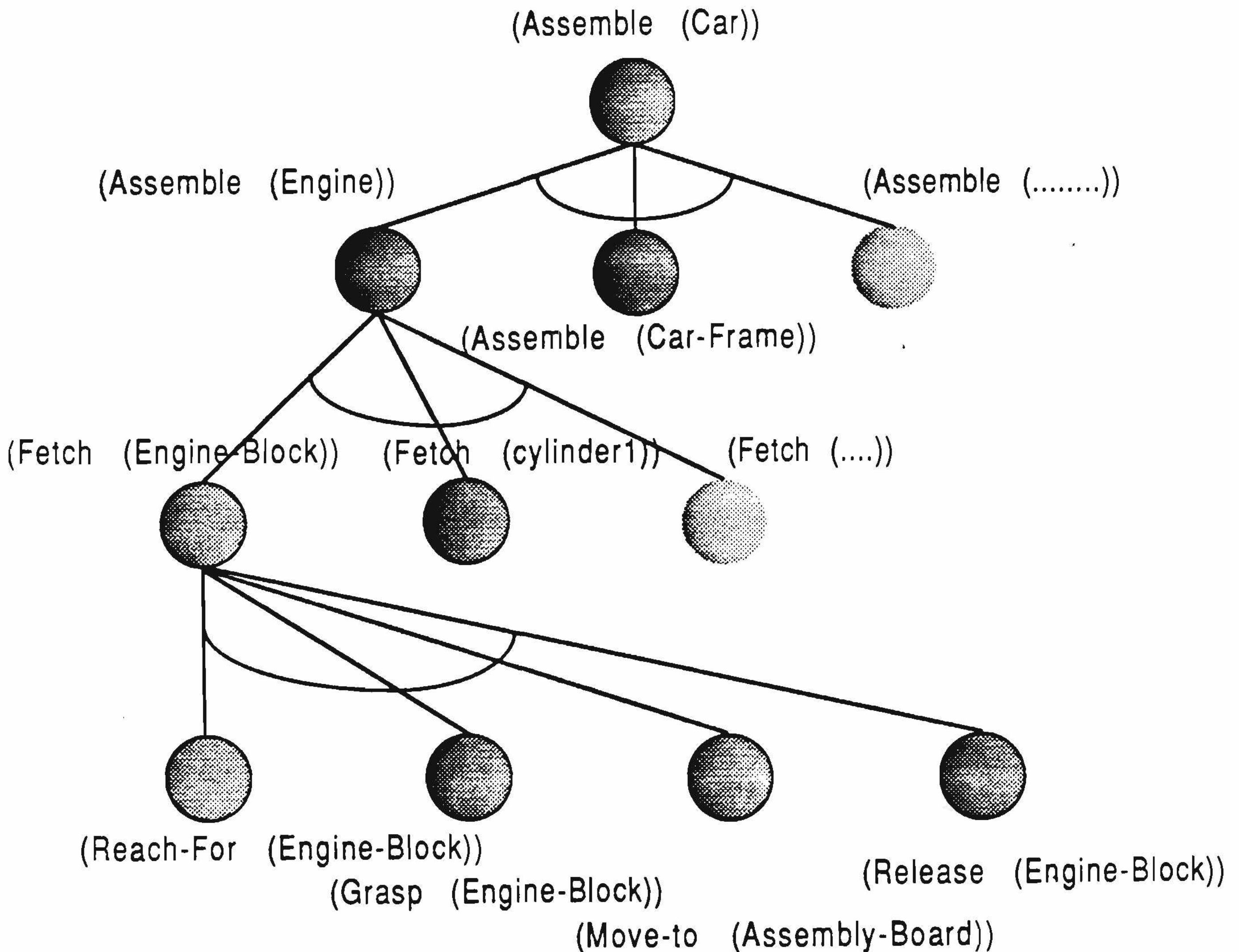

Figure 2-2: Hierarchy of Goals and Subgoals

directly comparable to the goal states neurons in the brain can be in. The main difference is that each goal/subgoal (neuron) of the brain is constantly active, monitoring its environmental input, while the goals/subgoals of the CAOS control system are passive until a particular goal state is wanted which involves activating neuroschemata with these goals/subgoals.

Each node and leaf has slots for storing information about the syntax of the goal, the arguments of the goal, the expected type and range of pre-inputs and post-inputs, the average run time to achieve the goal, the probability of success of obtaining the goal, and an output function. Figure 2-3 shows the node and leaf structures.

Information found in these node and leaf goals are specified by the user. This is similar to specifying

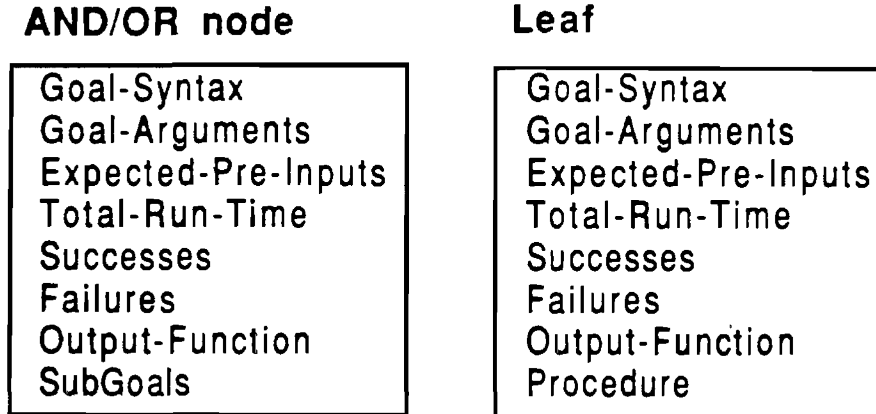

Figure 2-3: Structure of Node and Leaf Objects/Nodes

rules in an expert system. The *goal-syntax* specifies the goal state and the pre-input (if any) needed to obtain this goal state.

The goal arguments are simply the pre-input that the goal was activated with (needed input for achieving the goal state). *Expected pre-inputs* are functions (if any) which are used for checking the range and type of the pre-input that the goal was activated with. The *total run time* is the run time that the goal uses to achieve the goal state. *Successes* and *failures* are just what they say, the total number of successes and failures in trying to obtain the goal state. The *output function* is used to produce the output of the goal which implies that the wanted goal state has been achieved. Finally, the *subgoals* are the goals needed to be obtained for achieving the wanted goal state. This is usually referred to as goal/problem reduction. In the case of a leaf the subgoals would have been replaced with a single function.

2.4 The Global Data Base

The data base contains the known set of facts associated with a particular domain and is the model of the "world". The database used in the C implementation of CAOS was very simple, only containing information about polyhedral objects. Figure 2-4 shows an example of the information stored in the data base.

With each polyhedral object known to the data base, its three-dimensional CAGD model and current position and orientation in the environment are stored. In the LISP version of CAOS, *frames* will be used to represent information known to the system.

OBJECT1
3D_CAD_MODEL POSITION (X,Y,Z) ORIENTATION

OBJECT2
3D_CAD_MODEL POSITION (X,Y,Z) ORIENTATION

Figure 2-4: Data Base Objects

3. Exploiting Parallelism in CAOS

3.1 Introduction

The serial C version of the robot control system was partially transported from a VAX 11/780 to a Butterfly Parallel Processor [25]. The Butterfly is a multiple instruction, multiple data (MIMD) machine, and is connected to a host machine, a VAX 11/780. The Butterfly may have up to 256 processor nodes interconnected by a switching network called the *Butterfly Switch.* Each processor node has a co-processor called the Processor Node Controller (PNC) which is responsible for all memory references and transfers. The local Butterfly at The University of Utah has 19 Motorola MC68020 processor nodes, each having a Motorola MC68881 co-processor and 1 Mbyte of memory, except two, which have 4 Mbyte of memory. The processors operate at 16 MHz, due to a frequency doubler. References over the Butterfly Switch, to remote memory, usually takes about 4 microseconds round trip.

All code for the Butterfly is developed and compiled on the host machine. The executable code is then downloaded to the Butterfly, where it is run. There are two approaches we use to program the Butterfly: *Chrysalis* functions [21] and *Uniform System* [26] functions.

Each processor runs one copy of the operating system Chrysalis. This operating system is mainly written in C and supports communication and synchronization between processes running on different processors. This is done by means of dual queues which pass messages between these processes, and an event mechanism (similar to *signals* in UNIX). Chrysalis does not provide automatic resource allocation, load balancing or process migration, however [9]. Each user-developed program has to set up the data, create all necessary processes, and decide on which node(s) they will run. Five analogs to UNIX's seek-, open-, close-, read-, and write-functions enable access to files residing on the host machine.

Compared to Chrysalis, the Uniform System approach to programming the Butterfly provides the user with easier resource management. The Uniform System is built on top of Chrysalis and consists of several subroutines which take care of, for example, allocation of memory and processors, and generation of new tasks (processes). The user does not allocate memory space or processors explicitly, since the Uniform System takes care of the distribution of tasks on processors and provides special memory allocation routines. The Uniform System is especially suitable for homogeneous problems often found in low level computer vision programs.

3.2 Parallelism in the Human Brain

Information (input) to the brain arrives through a number of different pathways: Sound, Sight, Smell, Taste, and Touch Theses are senses that reach the conscious part of the brain. Examples of subconscious senses for joints and muscles are: Position, Force, Velocity, and Motion. These conscious and subconscious senses provide the brain with a constant inflow of information. They are all monitored and controlled by different segments of the brain in parallel. Each monitoring segment has millions of neurons where each neuron operates in parallel. The ability for the brain to process and control the complex human body as reliable and fast as it does relies heavily on the neurons' ability to operate in parallel. Even though the computer is able to operate much faster than a neuron when computing functions and relaying the result, no single processor can compete with the processing capabilities of the brain. Parallel processing is required if simulation of the brain, even only a simulation of a fractional part of it, is to be reality.

3.3 Parallelism in the Control System

Exploiting parallelism in the control system involves activating several neuroschemata on different processors, requiring complex communication and synchronization between various processes. This is implemented using Chrysalis. The control system uses the neuroschemata in the hierarchically organized tree, described earlier , to decide if subgoals can be started up in parallel. This occurs when different alternative subgoals can achieve the same goal with approximately the same expectation of success. In addition, subgoals can be started up in parallel when all needed inputs are provided, and any use of end effectors will not result in conflicts.

One of the advantages of using multiple processors to simultaneously execute alternative goal paths, is to prevent time delay due to an alternative's failure to obtain the goal. If one of the alternatives fails, or the results are not satisfactory, the result of another can be used instead. If the alternatives were not executed in parallel, and the most promising one failed, it would take longer to achieve a goal; the next alternative would be executed only after the first had failed.

When the hierarchical control allows parallelism, the parent neuroschema has to check if there are any processors available on which to start up "child processes" (new neuroschemata). If this is the case, the parent must also set up all the necessary data on the respective processors before it can initiate any child processes. The parent and child communicate using a dual queue, on which messages are posted. When a child is done, a special message informs the parent [6]. If no processor is available, however, the child

process must be started up on the same processor as the parent. Moreover, if there is only one way of obtaining a goal, the child will always be started on the same processor node as the parent, since there are no alternatives which can be started up in parallel.

The possibility of executing several alternative or independent neuroschemata simultaneously, can speed up the system considerably compared to executing it on a uniprocessor. How much faster it will actually run, depends on how well parallelism can be exploited in each particular case.

3.4 Parallelism in Programs

In addition to parallelism in the control system, inherent parallelism can be exploited in programs such as low level image processing. These programs deal with image data which requires extensive and time consuming operations. Implementing such programs has no complex control aspects because the processing is homogeneous, enabling the processors to run the same code on different data. This implies that the Uniform System is the best programming approach. One example is edge detection. In this case, the data (the image) can be split into several "chunks" and put onto the available processors, which all run the same edge detection program on their part of the image [26] (a homogeneous problem). There is no complex control aspects involved, like starting up different programs on different processors and taking care of dual queues for message passing between the processes.

3.5 Processor Utilization

The two categories of parallelism in the system, discussed above, could use as many parallel processors as there are possible processes. However, there is a limit on the number of processors, 19 in our case, and therefore the problem of processor utilization arises. There has to be a balance between the number of processors the two categories are allowed to occupy. Obviously, the most time consuming processes should use the maximum number of processors, thus reducing the number of "bottle necks" in the system. Since programs such as low level image analysis will be the most expensive part with regard to execution time, it is preferable that these processes occupy most of the processors on the Butterfly, so as to prevent unnecessary serial execution. The total execution time for achieving a goal will then be minimized. If the hierarchical control programs occupy just a few nodes, this will not hurt the overall performance significantly, since even the serial version of the control does not take much execution time.

4. CAOS Versus Expert Systems

The difference between expert systems and programs are that the knowledge in an expert system is a separate entity while in programs it is implicit. Moreover, expert systems manipulate knowledge while programs manipulate data. Expert systems organize its knowledge in three levels:

- Data
- Knowledge
- Control

A system organized in this way is said to be a *knowledge based system.* An expert system is a knowledge based system but the opposite is not necessarily true. An expert system is capable of learning from its errors. In contrast to *knowledge-based systems* an expert system must also be capable of *explaining* its behavior and decisions to the user. The structure of an expert system is shown in Figure 4-1.

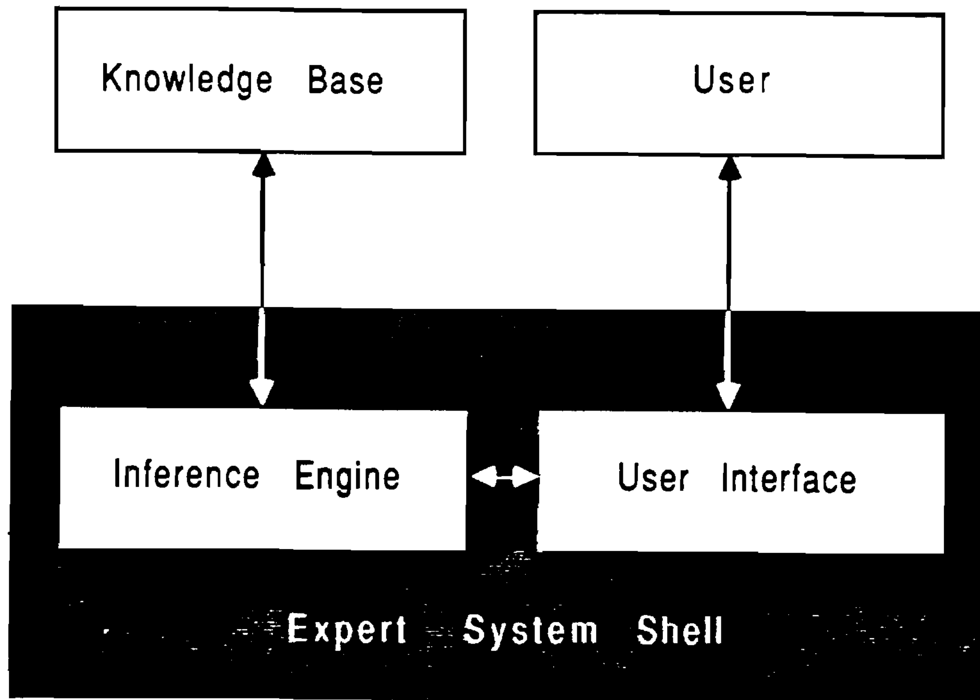

Figure 4-1: The Structure of an Expert System

There are many ways of representing knowledge. Some of the most used are *predicate logic, frames, and semantic nets* [5]. The knowledge for achieving a goal (solving a problem) in an expert system comes from the knowledge it possesses and not only form the formalism and inference schemes it employs.

In developing CAOS it was important that it had similarities to the human brain and also knowledge

based (expert) systems [13, 15, 23, 28]. The similarities to the brain are evident in that CAOS has basic building blocks (neuroschemata), is controlled in a hierarchical manner, is goal driven, and exploits parallelism if available. As in an expert system, CAOS has a knowledge base and a data base with facts and rules (rules for obtaining goals). Furthermore, like the inference engine (interpreter and scheduler), our neuroschemata contain the general problem solving knowledge. Moreover, in order to do anything "intelligent", both expert systems and our control system need a domain expert to provide them with knowledge on how to obtain goals.

The neuroschemata in CAOS provide a consistent way of interpreting how goals should be obtained. They use metarules in the form of probabilities and average run time when deciding which subgoals to pursue to achieve a main goal. Making CAOS goal driven (backward chaining) prevents problems with generating too many hypotheses, but does not avoid the problem of restricting the hypotheses to too narrow a range. In both systems the knowledge is permanent and consistent, and can be goal directed.

One main difference between CAOS and expert systems in general, is that our system can easily receive sensory information from the environment provided programs exist in its knowledge base enabling interaction with sensors. Implementing the system in LISP or C and using the neuroschemata as the basic building blocks, enable us to easily acquire this sensory information, and also control end effectors. In addition, the system can more easily be exported to the BBN Butterfly Parallel Processor than a system written using expert system tools, since this machine currently support only C and soon will support LISP.

The conventional expert system relies on symbolic information provided by the user to achieve a goal, and usually cannot interact with the environment through sensors. This seems to be the most important difference. But this does not mean that an expert system with the right expert-system-building tool could not provide this capability. We conclude that although CAOS is similar to expert systems, essential differences are found in the use of basic building blocks, neuroschemata, and easy sensor integration due to implementation in LISP and C.

5. An Overview of CAOS

5.1 Introduction

This chapter reviews the basic mechanisms of CAOS through examples. All currently working features of CAOS is presented. As describe earlier, CAOS is an action oriented hierarchical control system, a programming language for goal achievement. It is a language designed for enhanced human control interaction. CAOS is specially well suited for representing high level goals for systems such as robots. Knowledge about goals which CAOS is to achieve is provided by the user through *clauses*. A clause is defined as a list with a *head* (car of the list) and a *body* (cdr of the list).

(head body)

The head of the clause specifies a goal and its input. The body specifies how to obtain this goal.

((put-on (a b)) (and (get-space (a b)) (put-at (a s1))))

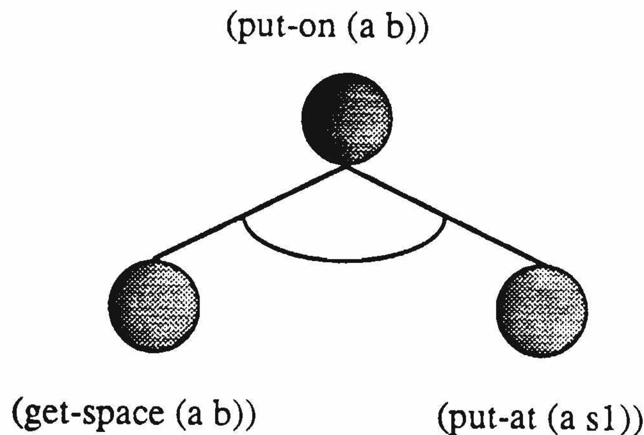

Figure 5-1: Goal: (put-on (a b))

In the above example, shown in Figure 5-1, the head is *(put-on (a b))*. The goal is to put an object *a* on top of another object *b*. To obtain this goal, the body has to be satisfied:

(and (get-space (a b)) (put-at (a s1)))

The information in the body specifies that the subgoal *get-space* AND the subgoal *put-at* both must be achieved if the main goal, *put-on*, is to be satisfied. In completing the knowledge for how to achieve the goal *(put-on (a b))* the following clauses are added.

```
((get-space (a b))
  (or
   (find-space (a b))
   (make-space (s1 a b))
   ))

((find-space (a b))
  (function
   (lambda (a b) (list 10.0 0.0 22.3))
   ))

((make-space (d a b))
  (and
   (get-rid-of (d))
   (find-space (a b))
   ))

((get-rid-of (d))
  (and
   (find-space (d d))
   (put-at (d s1))
   ))

((put-at (a b))
  (and
   (grasp (a))
   (move-object (a b))
   (ungrasp (a))
   ))

((grasp (a))
  (function
   (lambda (a) 'grasped))
   )

((move-object (a b))
  (function
   (lambda (a b) 'moved))
   )

((ungrasp (a))
  (function
   (lambda (a) 'ungrasped))
   )
```

As one can see, two new clause types appear in the above program. These are the *or* and *function* clauses. The *or* clause has the same syntax as the *and* clause, but with the symbol *or* as the first element of the body. An example is shown in Figure 5-2.

((get-space (a b)) (or (find-space (a b)) (make-space (s1 a b))))

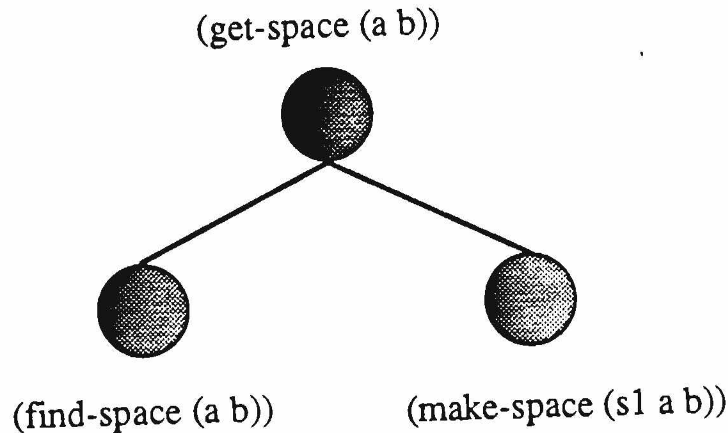

Figure 5-2: Goal: (get-space (a b))

To achieve the goal of getting some space for *a* on top of *b*, the subgoal *find-space* OR the subgoal *make-space* must be achieved. Which goal to pursue first is taken care of by CAOS. It uses previous experience about the subgoals when considering which one has the best chance of success, as described earlier.

The other new clause type that was presented in the above program was the *function* type, shown in Figure 5-3.

((find-space (a b)) (function (lambda (a b) (list 10.0 0.0 22.3))))

The meaning of this clause is that to obtain the goal of finding some space for *a* on top of *b* the function expressed in the body has to be executed (applied to the given arguments). The function can be any LISP lambda expression. Refer to [18] for an introduction to LISP.

(find-space (a b))

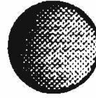

Figure 5-3: Goal: (find-space (a b))

The above are some simple goals written in CAOS clauses and we are ready to test them out. But, before we do that we must let CAOS now about the goals. This is done using the command *(consult clauses)*. For the above example the following command would be given. The resulting knowledge of CAOS can be depicted as in Figure 5-4.

```
(consult '(

((put-on (a b))
  (and
   (get-space (a b))
   (put-at (a s1))
   ))

((get-space (a b))
  (or
   (find-space (a b))
   (make-space (s1 a b))
   ))

((find-space (a b))
  (function
   (lambda (a b) (list 10.0 0.0 22.3))
   ))

((make-space (d a b))
  (and
   (get-rid-of (d))
   (find-space (a b))
   ))
```

```
((get-rid-of (d))
  (and
   (find-space (d d))
   (put-at (d s1))
   ))

((put-at (a b))
  (and
   (grasp (a))
   (move-object (a b))
   (ungrasp (a))
   ))

((grasp (a))
  (function
   (lambda (a) 'grasped))
   )

((move-object (a b))
  (function
   (lambda (a b) 'moved))
   )

((ungrasp (a))
  (function
   (lambda (a) 'ungrasped))
   )
))
```

After executing this command, CAOS will respond with "Clauses consulted", which mean that the program is loaded into the global knowledge base. Finally, we can now test some of the goals. This is done by using the command *(achieve goal)*. Achieve tells CAOS that it should try to achieve the goal given to it. If we now gave the following command to CAOS

(achieve '(put-on (red-cube blue-cube)))

the result would be

UNGRASPED

So what exactly happened here ? A tracing of the achievement of the goal is shown below and should make it clearer. The numbers on the left hand side of each trace statement indicates their order in the

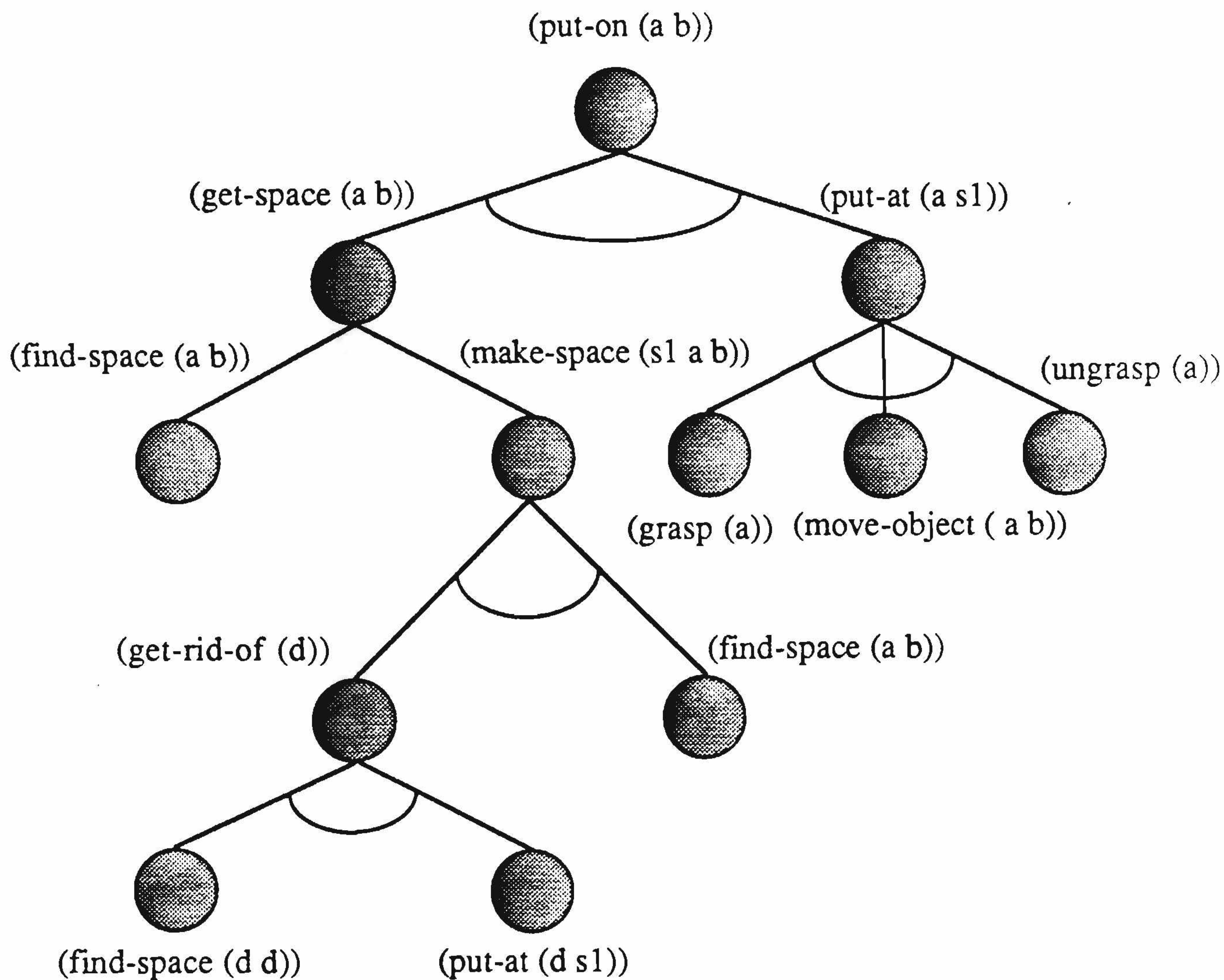

Figure 5-4: CAOS's Knowledge Base After The Consult Command

sequential execution.

```
1  Activating: (put-on (red-cube blue-cube))
2  Activating: (get-space (red-cube blue-cube))
3  Activating: (find-space (red-cube blue-cube))
4  Returning : (find-space (red-cube blue-cube))
               -> (10.0 0.0 22.3)
5  Activating: (put-at (red-cube (10.0 0.0 22.3)))
6  Activating: (grasp (red-cube))
7  Returning : (grasp (red-cube))
               -> grasped
8  Activating: (move-object (red-cube 10.0 0.0 22.3))
```

```
9  Returning : (move-object (red-cube 10.0 0.0 22.3))
                -> moved
10 Activating: (ungrasp (red-cube))
11 Returning : (ungrasp (red-cube))
                -> ungrasped
12 Returning : (put-at (red-cube (10.0 0.0 22.3)))
                -> grasped
13 Returning : (put-on (red-cube blue-cube))
                -> Ungrasped
```

5.2 Syntax and Meaning of CAOS clauses

This section gives a systematic treatment of the syntax and semantics of CAOS clauses. Topics included are:

- AND clauses
- OR clauses
- FUNCTION clauses
- Expected PRE- and POST-INPUT clauses
- OUTPUT clauses

5.2.1 AND clauses

As seen in section one, an AND clause has the following syntax

(head (AND subgoals))

This clause type represents an AND node. Figure 5-5 shows an example. The goal (the head of the clause) is satisfied if and only if all of its subgoals are satisfied. The subgoals are processed left to right. The result of an AND node is the result of the last subgoal, if nothing else is specified (see *output clauses*). A goal fails (is not satisfied) when the result is *nil.* Any other result implies success. Note that the default result of an AND node is identical with the result of an *and* in LISP.

5.2.2 OR clauses

An OR clause is similar in syntax to the AND clause.

(head (OR subgoals))

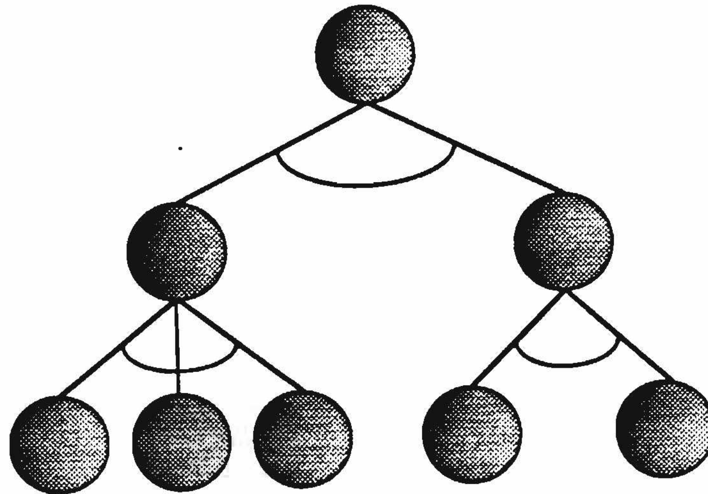

Figure 5-5: AND nodes

The OR clause represents an OR node, as shown in Figure 5-6. The goal (the head of the body) is satisfied if any one of the subgoals are satisfied. The result of the OR node would then simply be the first subgoal to be satisfied if nothing else where specified (see *output clauses*), which is the default.

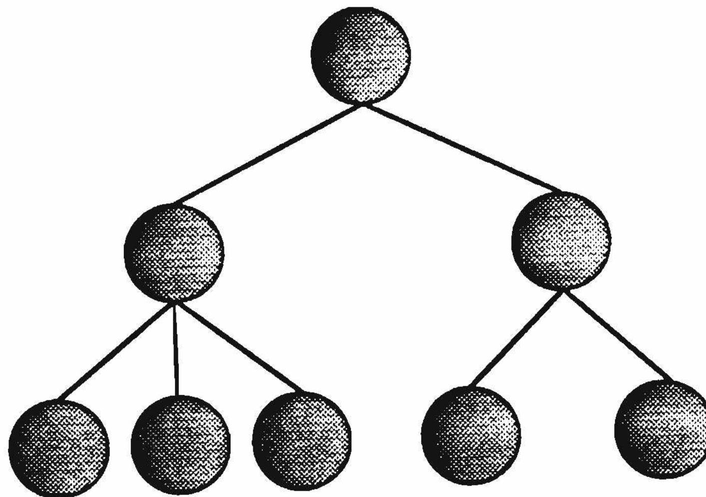

Figure 5-6: OR nodes

Which subgoal to try to achieve first (which path to follow in the tree) is taken care of by the CAOS system, or more specifically the neuroschema controlling the goal. It uses the average run time of the subgoal (run time is only registered if the goal succeeded) and its probability of success to compute the

expectation of success for that subgoal as explained in detail in earlier chapters.

The subgoals are processed in sequential order, starting with the smallest E(s) (indicating best chance of success), until one subgoal succeeds. If none of the subgoals succeed, the main goal of the OR node will fail.

5.2.3 Function clauses

The *function clauses* represent the leafs of a tree, as seen in Figure 5-7. The syntax of this clause type is as follows:

((goal (input args)) (function (lambda (input args) function-body)))

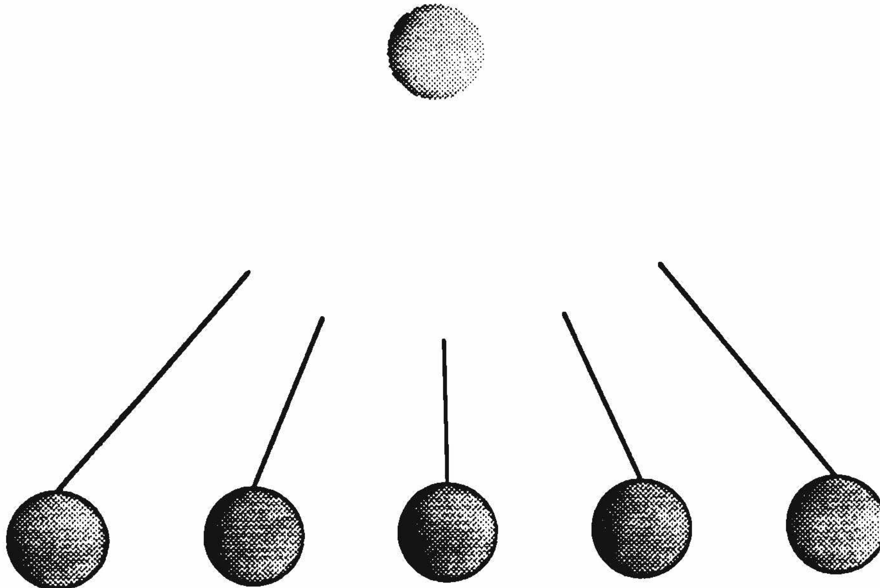

Figure 5-7: Leafs

The result of the goal in this clause type is the result of applying the function to the given arguments. This is the default result if nothing else is specified (see *output clauses*). As one can see, the function is nothing but a lambda expression in LISP, and hence, the function clause can be any lambda function expressible in LISP.

5.2.4 Expected pre- and post-inputs clauses

Pre- and post-input clauses enable us to specify what domain the input should be in. Pre-input is the input given as information to the goal when it is activated (it is the cdr of the head).

```
(put-on (red-cube blue-cube))
```

In the above example, *(red-cube blue-cube)* is the pre-input. Post-input is the result of the subgoals of the goal. Pre- and post-input is the total input to the goal and acts as the input to the *output function* of the goal (see *output clauses*). The syntax of the pre- and post-input clauses are similar.

```
((put-on (a b)) (a (function (lambda (a) (objectp a)))))
((put-on (a b)) (b (function (lambda (b) (objectp b)))))
((put-on (a b)) (s1 (function (lambda (s1) (pointp s1)))))
```

In the above example we have specified that both *a* and *b*, the pre-input to the goal, must be of the type object. If any of them fails this requirement, the goal fails immediately. I also specified that the result of subgoal one (s1; the result of subgoal two would be s2 etc.), the post-input to the goal, must be a point. If this is not true, then the goal also fails.

5.2.5 Output clauses

This clause type enables the user to specify a function that will use the pre- and post-input to return the output/result of the goal. If no output function is specified for a goal then the default output is used as discussed earlier. The syntax (as a LISP example) is as follows:

```
((put-on (a b))
   (output
    (function (lambda (a b s1)
      (format t "Placed ~A on ~B at point ~A~%" a b s1)))))
```

5.3 CAOS Commands

In this section we will go through the commands which the user has available when using CAOS. The following commands are discussed:

- Achieve
- Consult
- display
- erase
- help
- reconsult
- trace-on
- trace-off

5.3.1 Achieve

Syntax: (achieve '(goal-name (input1 input2 ...)))

With this command the user can achieve a goal specified in the goal base. The result of this command will be either *nil* (indicating that the goal failed) or any other result when the goal succeeds. To add new goals to the goal base, use the commands *(consult clauses)* or *(reconsult clauses).*

5.3.2 Consult

Syntax: (consult clauses)

With this command the user can add new goals to the goal base. This is done by making a list of goals expressed as CAOS clauses, and then consulting them. Consult will not modify a goal if the same goal is defined in more than one clause. Refer to *(reconsult clauses)* for this.

5.3.3 Display

Syntax: (display) or (display goal-name)

With this command the user can display the information stored in the goal base for all the goals, or for one particular goal. The information shown is the average run time of the goal, failures and successes,

goal syntax, and input and output information.

5.3.4 Erase

Syntax: (erase) or (erase goal-name)

With this command the user can erase all the goals in the goal base or only one particular goal.

5.3.5 Help

Syntax: (help) or (help command)

With this command the user can get short help on all the commands at one time, or more extensive help on a particular command.

5.3.6 Reconsult

Syntax: (reconsult clauses)

Same as consult, but will modify a goal if defined more than once in a goal listing, or if consulted previously.

5.3.7 Trace-on

Syntax: (trace-on)

With this command, CAOS will print out messages concerning what goal(s) has (have) been activated when you are using the achieve command. It will also print out messages about which goal is being consulted when using the consult functions.

5.3.8 Trace-off

Syntax: (trace-off)

Turns the trace mode off (turned on by trace-on).

5.4 Example: The Cube of a Number

This example shows how CAOS works and how the user can program in a goal directed way with CAOS. Five different ways of computing the cube of a number is presented.

(cube-1 (x)) is a goal (a leaf) that can not be divided down further, as shown in Figure 5-8. The cube is simply returned by a function call on the argument x. *(cube-2 (x))* returns the cube of x by dividing itself into one subgoal, (cube-1 (x)). *(cube-3 (x))* is a bit more complicated. It returns the cube of x by dividing itself into two subgoals in which both must be obtained. The result of the first subgoal , *(mult (x x))*, is **s1 = x*x**. The result of *(mult (s1 x))* is **s2 = s1 * x**, which is also the result of cube-3, the main goal. *(cube-4 (x))* is a bit funny. It only returns the cube of x if it is either raining *or* blowing. The result of the goal in this example is returned using the option of specifying an output function for the goal. The result in this case is **x*x*x**, as specified in the output function of cube4. *(cube-5 (x))* is another version of cube-4, but the cube of x will only be returned if the *pre-input x is > 0* and the *post-input*, the result of cube-4 in this case, *is < 1000*. If these conditions are met, the result is formated using the output function of the main goal cube-5, in this case: X cubed = 'x*x*x'. The following listing and Figure 5-8 shows the goals discussed above.

```
(consult '(
   ((cube-1 (x))
    (function (lambda (x) (* x x x))))

   ((cube-2 (x))
    (and
     (cube-1 (x))
     ))

   ((cube-3 (x))
    (and
     (mult (x x))
     (mult (s1 x))
     ))

   ((mult (x y))
    (function (lambda (x y) (* x y))))

   ((cube-4 (x))
    (or
     (raining-? ())
     (blowing-? ())
     ))

   ((cube-4 (x))
    (output (function (lambda (x) (* x x x)))))
```

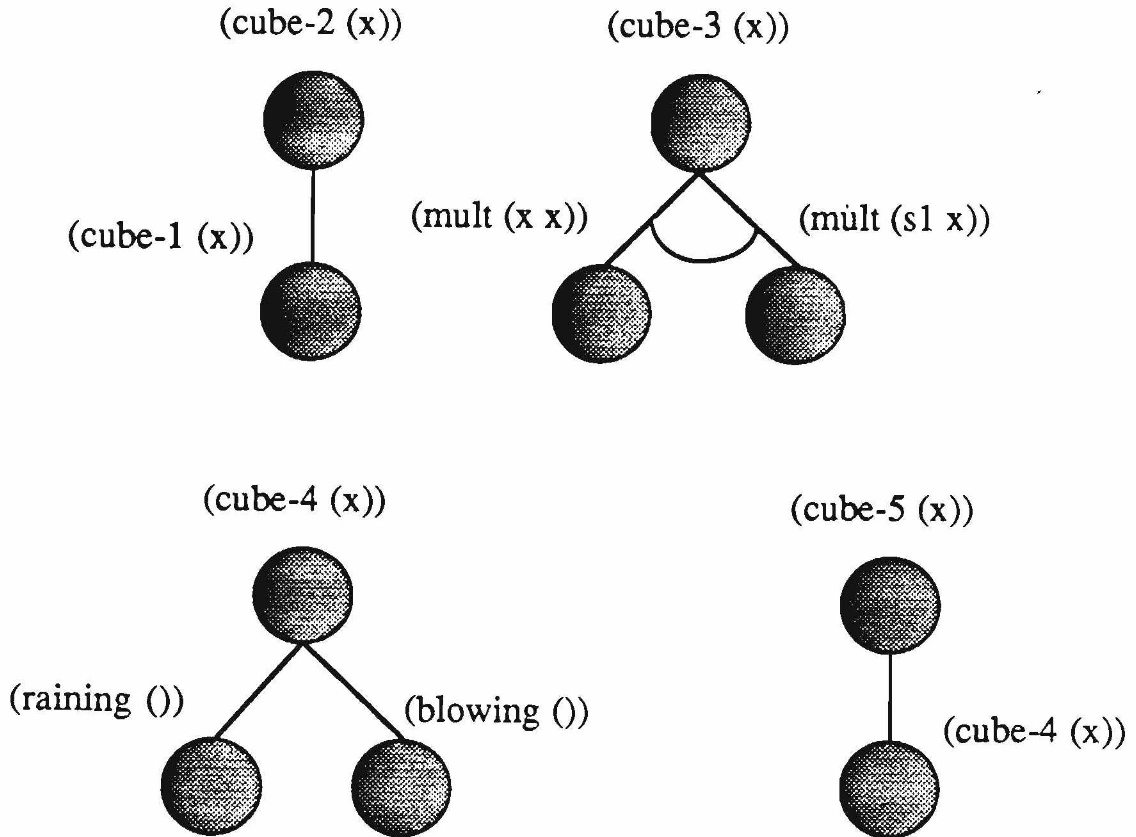

Figure 5-8: Different ways of obtaining the goal, (cube (x))

```
((raining-? ())
 (function (lambda () nil)))

((blowing-? ())
 (function (lambda () T)))

((cube-5 (x))
 (and
  (cube-4 (x))))

((cube-5 (x))
 (x (function (lambda (x) (> x 0)))))

((cube-5 (x))
 (s1 (function (lambda (s1) (< s1 1000)))))
```

```
((cube-5 (x))
 (output
  (function
   (lambda
    (s1) (format t "X cubed = ~A" s1))))
))
```

5.5 Example: Graphics Demo

This example shows some graphics routines (written using Starbase) which was intended for showing a robot moving in a scene (path planning and object recognition). Due to limited time, we was only able to do a simple graphics interface, leaving the path planning for later. This example displays two windows on the screen. In window 2 (the smallest) the scene is shown from the front. In window 1 the scene is repeatedly shown at different viewing angles, starting with front view and then moving up, to the left and into the scene. All the necessary routines for actually simulating a robot moving in the room, looking through its camera, is presented here.

```
(consult '(
   ((show (xp yp d))
    (and
     (set-up-graphics-device ())
     (draw-scene-in-w2 ())
     (draw-scene-in-w1 ())
     (show-time (xp yp d))
     (close-graphics ())
     ))

   ((draw-scene-in-w1 ())
    (and
     (current-window-1 ())
     (draw-scene ())
     ))

   ((draw-scene-in-w2 ())
    (and
     (current-window-2 ())
     (draw-scene ())
     ))

   ((show-time (xp yp d))
    (and
     (not-done-? (d))
     (fun-draw (xp yp s1))
     ))

   ((not-done-? (d))
```

```
 (function
  (lambda (d) (if (< d 10) (+ 1 d) nil))))

((fun-draw (xp yp d))
 (and
  (get-perspective-transformation ())
  (set-transformation-to (s1))
  (modify-x (xp))
  (modify-y (yp))
  (get-transformation (s3 s4 d))
  (set-transformation-to (s5))
  (gclear ())
  (draw-scene ())
  (pop-matrix ())
  (pop-matrix ())
  (show-time (s3 s4 d))
  ))

((modify-x (x))
 (function (lambda (x) (- x 0.02))))

((modify-y (y))
 (function (lambda (y) (- y 0.015))))

((close-graphics ())
 (function (lambda () (gend))))

((gclear ())
 (function (lambda () (gclear))))

((set-up-graphics-device ())
 (function
  (lambda () (set-up-graphics-device))))

((pop-matrix ())
 (function
  (lambda () (pop-transformation-matrix))))

((get-transformation (xp yp d))
 (function
  (lambda (xp yp d)
   (get-transformation xp yp d))))

((get-perspective-transformation ())
 (function
  (lambda () (get-perspective-transformation))))

((draw-scene ())
 (function
  (lambda () (draw-scene))))

((set-transformation-to (m))
 (function
```

```
    (lambda (m) (set-transformation-to m))))

  ((current-window-1 ())
   (function
    (lambda () (current-window-scene))))

  ((current-window-2 ())
   (function
    (lambda () (current-window-camera))))
  ))
```

For more information about CAOS, a complete source code listing in both C and LISP are presented in the appendices.

6. Conclusions and Future Work

This thesis is divided into five chapters, a preface, this conclusion, and two appendices. In chapter one we discussed, compared, and contrasted neurons, schemata, and neuroschemata as basic building blocks in control systems. Chapter two presented the theoretical aspects of CAOS, and chapter three elaborated on parallelism in the system. A comparison to knowledge based system was discussed in chapter four. Finally, an overview of CAOS and its user interface was presented in chapter five, with references to appendix A and B for the complete CAOS C and LISP source code.

CAOS, an approach to robot control, was developed with three of the important aspects of human intelligence in mind: basic building blocks, hierarchical organization, and parallel processing. The system consists of three basic parts which include the knowledge base, the data base, and the scheduler for hierarchical control. This scheduler and the neuroschemata use information found in the knowledge base and data base to determine how to obtain a goal given by the user. Meta rules in the form of probability of success and average run time is used in the system for planning.

Examples showing how the system functions was presented, including a simulation of object manipulation by a robot.

Current software and hardware technologies enables limited use of parallel processing, utilized in CAOS, but future work on this system will include drastic changes in control structure and knowledge representation. Neuroschemata, with some modifications, will still function as the basic units of the control system, while the control structure will change from being strictly hierarchical to be spherical [27].

We conclude that although CAOS is similar to expert systems in some respects, they are set apart by differences such as using neuroschemata as a basic unit, the easy intergration of sensory input and output, planning using meta knowledge, and its learning capabilities.

Future work will consist of completing a parallel implementation of CAOS and obtaining results for speedup when comparing parallel versus serial processing. We will also study processor utilization and process synchronization. The control structure of CAOS will also be restructured to include both hierarchical and heterarchical control. Finally, we will continue to expand the scope of tasks which the control system can handle, including recognition of more complex 3-D objects and their assembly.

7. Appendix A

CAOS Source Code: LISP Version

This appendix lists the complete LISP source code of CAOS.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File    : compcaos.l
; Author  : Nils Thune
; Created : December 12, 1986
; Mode    : Common Lisp
;
; Purpose : Compilation file for CAOS.
;
; Copyright (c) 1986, Nils Thune.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(load "/net/ug/u/grads/shebs/objects")

(format t "Compiling CAOS node objetcs . . . . . ~%")
(compile-file "/net/cs/u/class/u-nthune/proj86/LISP/nodes.l")
(load "/net/cs/u/class/u-nthune/proj86/LISP/nodes")

(format t "Compiling CAOS globals . . . . . . . .~%")
(compile-file "/net/cs/u/class/u-nthune/proj86/LISP/globals.l")
(load "/net/cs/u/class/u-nthune/proj86/LISP/globals")

(format t "Compiling CAOS consult functions . . . . . .~%")
(compile-file "/net/cs/u/class/u-nthune/proj86/LISP/consult.l")

(format t "Compiling CAOS tools . . . . . . . . . . . .~%")
(compile-file "/net/cs/u/class/u-nthune/proj86/LISP/tools.l")

(format t "Compiling CAOS help functions .. . . . . . .~%")
(compile-file "/net/cs/u/class/u-nthune/proj86/LISP/help.l")

(format t "Compiling CAOS achieve functions .. . . . . . .~%")
(compile-file "/net/cs/u/class/u-nthune/proj86/LISP/achieve.l")

(format t "Compiling CAOS graphic routines .. . . . . . . .~%")
(compile-file "/net/cs/u/class/u-nthune/proj86/LISP/graph-rout.l")

(format t "Compiling CAOS graphic routines .. . . . . . . .~%")
(compile-file "/net/cs/u/class/u-nthune/proj86/LISP/scene.l")

(format t "Compiled CAOS system !!!")
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File    : caos.l
; Author  : Nils Thune
; Created : November 17, 1986
;
; Copyright  (c) 1986, Nils Thune.
;
; Purpose :
;
; CAOS is a hierarchical action oriented control system.  The system builds
; a hierarchical tree with 2 node types (OR and AND) and 1 leaf type
; (PROGRAM), where the root of the tree is the main goal to be achieved.
;
; The 2 node types and the leaf type (LISP objects) represents goals or
; subgoals in the tree.  Each node and leaf type (goal types) have a method
; that controls the achievement of that particular goal (node or leaf).
;
; One node, or leaf, with its control method is called a "neuroschema",
; named after its similarities with neurons and schemas.
;
; The neuroschemas interacts with a goal base to achieve a particular
; goal.  This goal base contains nodes with information about a particular
; goal (node or leaf type goal).
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(format t "Loading LISP objects . . . . .~%")
(load "/net/ug/u/grads/shebs/objects")

(format t "Loading CAOS globals . . . . .~%")
(load "/net/cs/u/class/u-nthune/proj86/LISP/globals")

(format t "Loading CAOS node objects . . . . .~%")
(load "/net/cs/u/class/u-nthune/proj86/LISP/nodes")

(format t "Loading CAOS consult functions . . . . .~%")
(load "/net/cs/u/class/u-nthune/proj86/LISP/consult")

(format t "Loading CAOS tools . . . . .~%")
(load "/net/cs/u/class/u-nthune/proj86/LISP/tools")

(format t "Loading CAOS help functions . . . . .~%")
(load "/net/cs/u/class/u-nthune/proj86/LISP/help")

(format t "Loading CAOS achieve functions . . . . .~%")
(load "/net/cs/u/class/u-nthune/proj86/LISP/achieve")

(format t "Loading CAOS starbase graphics routines . . . . .~%")
(load "/net/cs/u/class/u-nthune/proj86/LISP/graph-rout")

(format t "Loading CAOS starbase scene drawing routines . . . . .~%")
(load "/net/cs/u/class/u-nthune/proj86/LISP/scene")

(format t "CAOS system loaded !!!~%")
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File    : globals.l
; Author  : Nils Thune
; Created : December 12, 1986
; Mode    : Common Lisp
;
; Purpose : All globals used in CAOS.
;
; Copyright (c) 1986, Nils Thune.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defvar *trace* nil)        ;; Trace mode initialized to OFF.

(defvar *goal-base* nil)    ;; Contains goals in form of an alist.
                            ;; ((goal-name node-object) (... ...) .... )

(defvar failed nil)         ;; Indicates that a goal failed (implies that
                            ;; a goal can not return nil as its success value.

(defvar *consult-mode* 1)   ;; Specifies the current consult mode, initialized
                            ;; to define mode.

(defvar *define* 1)         ;; This consult mode specifies that if a goal all-
                            ;; ready has been defined, it can not be redefined.

(defvar *redefine* 2)       ;; This consult mode specifies that a goal can be
                            ;; redefined.

;; Used when returning the result of a subgoal (the 'result' and 'sn' is listed
;; together).  The variable indicates that a goal can only have 9 subgoals,
;; but by adding (10 s10) etc., any number can be allowed.

(defvar *subgoal-variables*
  '((1 s1) (2 s2) (3 s3) (4 s4) (5 s5) (6 s6) (7 s7) (8 s8) (9 s9)))



;;; Next are some globals used in the graphics interface made for CAOS.

;; I am usually using 'device' and 'driver'.

(defvar device "/dev/graphics")
(defvar driver "hp98710")
(defvar device2 "/dev/screen/foo")
(defvar driver2 "hp300h")
(defvar device3 "/dev/graphics")
(defvar driver3 "hp98700")

(defconstant WX1 0.0)  ;; Lower left corner of scene in world coordinates.
(defconstant WY1 0.0)
(defconstant WZ1 0.0)
(defconstant WX2 10.0) ;; Upper right corner of scene in world coordinates.
(defconstant WY2 10.0)
(defconstant WZ2 -20.0)
```

```
(defvar *fd* 0)          ;; File descriptor for graphics device.

;; Open mode used when opening the graphics device.

(defvar *open-mode* (logior hp-ux_3g:THREE_D hp-ux_3g:INIT))

(defvar *xr* 0.0)       ;; view reference point.
(defvar *yr* 0.0)
(defvar *zr* 0.0)

(defvar *dxn* 0.0)      ;; view plane normal.
(defvar *dyn* 0.0)
(defvar *dzn* 1.0)

(defvar *dxup* 0.0)     ;; view-up direction.
(defvar *dyup* 1.0)
(defvar *dzup* 0.0)

(defvar *view-distance* 0.0)  ;; view distance.

;; Scaling matrix for the camera.

(setf *camera-scale* (make-array '(4 4) :initial-contents
                                 '((2.0 0.0 0.0 0.0)
                                   (0.0 2.0 0.0 0.0)
                                   (0.0 0.0 2.0 0.0)
                                   (0.0 0.0 0.0 1.0))))

;; The perspective transformation matrix used for the scene.

(setf -zp -15.0)
(setf yp 10.0)
(setf xp 5.0)
(setf *perspective-transformation* (make-array '(4 4) :initial-contents
                                               `((,-zp 0.0 0.0 0.0)
                                                 (0.0 ,-zp 0.0 0.0)
                                                 (,xp  ,yp 0.0 1.0)
                                                 (0.0  0.0 0.0 ,-zp))))


;;; Next follows help information for each command in CAOS stored in an
;;; a-list.

(defvar *help-list*
 '((achieve
    "Syntax: (achieve '(goal-name (input1 input2 ... )))
     With this command you can achieve a goal specified in the goal base.
     To find out which goals are available, use the command (display).
     To add new goals to the goal base use the commands (consult clauses)
     or (reconsult clauses).
     ")

   (consult
    "Syntax: (consult clauses)
     With this command you can add new goals to the goal base.  This is
```

```
      done by making a program consisting of clauses.  Refer to the user
      manual for the syntax and examples of clauses.  Consult will not
      modify a goal if the same goal is defined in more than one clause.
      Refer to (reconsult clauses) for this.
      ")

    (display
     "Syntax: (display) or (display goal-name)
      With this command you can display the information stored in the goal
      base for all the goals, or one particular.  The information shown is
      in particular the average run time of the goal, failures and successes
      , goal syntax, and input information.
      ")

    (erase
     "Syntax: (erase) or (erase goal-name)
      With this command you can erase all the goals in the goal base or only
      on particular goal.
      ")

    (help
      "Syntax: (help) or (help command)
      With this command you can get short help on all the commands at one
      time, or more extensive help on a particular command.
      ")

    (reconsult
     "Syntax: (reconsult clauses)
      Same as consult, but will modify a goal if defined more than once
      in a progam.
      ")

    (trace-on
     "Syntax: (trace-on)
      With this command, CAOS will print out messages conserning which
      goal(s) have been activated when you are using the achieve command.
      It will also print out messages about which goal is beeing consulted
      when using the consult functions.
      ")

    (trace-off
     "Syntax: (trace-off)
      Turns the trace mode off (turned on by trace-on).
      ")))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File    : nodes.l
; Author  : Nils Thune
; Created : December 12, 1986
; Mode    : Common Lisp
;
; Purpose : Definitions of objects used in CAOS.
;           - generic-goal
;           - or-node
;           - and-node
;           - program
;
; Copyright (c) 1986, Nils Thune.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; Below follows the definitions of goal types; there are 3 types of goals.
; The 3 types are built around a generic goal type.  The reason for using
; objects for goal types, is that the neuroscema (the controller) for these
; 3 types are different for each of them; using objects gives us the advantage
; of methods, hence the neuroschemas are implemented as 3 methods with the
; same name.

;;; GENERIC-GOAL: Has information common to all three goal types.  The three
;;; goal types inherits slots from this generic goal type.

(define-type generic-node
  (:var goal)                            ; Syntax of this goal
  (:var goal-args)                       ; Arguments in the goal.
  (:var expected-pre-inputs)             ; Expected type/range of pre-inputs.
  (:var expected-post-inputs)            ; Expected type/range of post-inputs.
  (:var total-time (:init 0))            ; Total run time to achieve this goal.
  (:var successes (:init 1))             ; # of successes obtaining this goal.
  (:var failures (:init 1))              ; # of failures.
  (:var output-func-args)                ; Arguments in the output function.
  (:var output-function)                 ; Function to compute the goal output
  :all-initable
  :all-settable
  :all-gettable
  )

;;; OR-NODE: Is a goal that exists as a goal node in the hierarchical control
;;; tree.  It has a slot for subgoals in addition to the slots inherited from
;;; the generic goal. An OR-NODE can only be a node in the control tree,
;;; never a leaf.

(define-type or-node                     ; An OR node goal type.
  (:inherit-from generic-node)
  (:var subgoals)                        ; The subgoals needed to obtain this goal.
  :all-initable
  :all-settable
  :all-gettable
  )

;;; AND-NODE: Is a goal that exists as a goal node in the hierarchical control
```

```
;;; tree.  It has a slot for subgoals in addition to the slots inherited from
;;; the generic goal. An AND-NODE can only be a node in the control tree,
;;; never a leaf.

(define-type and-node                    ; An AND node goal type.
  (:inherit-from generic-node)
  (:var subgoals)                        ; The subgoals needed to obtain this goal.
  :all-initable
  :all-settable
  :all-gettable
  )

;;; PROGRAM: Has a slot for a procedure (lambda exp) that it achieves
;;; the goal with. A PROGRAM is always a leaf in the control tree, never
;;; a node.

(define-type program                     ; A leaf in the hierarchical tree.
  (:inherit-from generic-node)
  (:var procedure)                       ; A lambda expression used to obtain the
  :all-initable                          ; goal of the leaf.
  :all-settable
  :all-gettable
  )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File    : consult.l
; Author  : Nils Thune
; Created : November 20, 1986
; Mode    : Common Lisp
;
; Purpose : Defines the functions CONSULT and RECONSULT which enables the
;           user to define goals.
;
; Copyright (c) 1986, Nils Thune.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;; CONSULT is one way of communicating our goals to CAOS.  Consult takes
;;; a list of clauses and adds them to the goal base.  Consult does not
;;; redefine a goal that has allready been defined.

(defun consult (clauses)
  (setf *consult-mode* *define*)
  (do-consult clauses))

;;; RECONSULT is another way of communicating our goals to CAOS.  Reconsult
;;; takes a list of clauses and adds them to the goal base.  Reconsult
;;; does redefine a goal that has allready been defined.

(defun reconsult (clauses)
  (setf *consult-mode* *redefine*)
  (do-consult clauses))

;;; For each clause found in the list CLAUSES, the clause is integrated into
;;; the goal base depending on its type.

(defun do-consult (clauses)
  (dolist (clause clauses (format t "~%Clauses consulted.~%"))
          (if *trace* (format t "Consulting: ~A~%" clause))
          (case (clause-type clause)
                ('and      (integrate-node 'and-node clause))
                ('or       (integrate-node 'or-node clause))
                ('function (integrate-node 'program clause))
                ('output   (integrate-output clause))
                (T         (integrate-input clause)))))

;;; Returns the first element in the clause body.

(defun clause-type (clause)
  (caadr clause))

;;; Adds a funtion to a particular input argument in the goal definition, which
;;; specifies the type of input to expect for that argument.  The function
;;; should return T or nil (specified by the user).

(defun integrate-input (clause)
  (let ((node (lookup-node (clause-name clause)))
        (symbol (clause-type clause))
        )
    (cond ((null node) (format t "~A~%Goal not defined. Ignoring it !!~%" clause
```

```
          ((post-input-function-? clause node (symbol-name symbol)))
          ((pre-input-function-? clause node symbol))
          (T (format t "~A~%" clause)
             (format t "Don't recognize the the first element of the body.~%")
             (format t "Ignoring it !!~%")))))

;;; Returns the node object associated with a named goal.

(defun lookup-node (name)
  (cadr (assoc name *goal-base*)))

;;; Returns the name of the goal in the given clause.

(defun clause-name (clause)
  (caar clause))

;;; Post input to a goal is the result of its subgoal(s).  A variable
;;; for post input is defined as sn, where n is the subgoal number.
;;; If sn is the first element of the clause body, then the rest is a function
;;; that specifies what the domain of sn should be (functions should return
;;; T or nil, specified by the user).

(defun post-input-function-? (clause node symbol)
  (if (and (equal 's (read-from-string symbol :start 0 :end 1))
           (numberp (read-from-string (subseq symbol 1))))
      (setf (expected-post-inputs node)
            (append (remove-instance (read-from-string symbol)
                                     (expected-post-inputs node))
                    (list (list (read-from-string symbol)
                                (eval (clause-function clause))))))))

;;; Returns the cdr of the clause body.

(defun clause-function (clause)
  (cadadr clause))

;;; Pre input to a goal is the input that the goal is called with. The
;;; argument names for pre inputs has no restriction.  If an argument
;;; name is the first element of the clause body, then the rest is a function
;;; that specifies what the domain of the argument should be (functions
;;; should return or nil, specified by the user).

(defun pre-input-function-? (clause node symbol)
  (dolist (current (goal-args node) nil)
          (cond ((equal symbol current)
                 (setf (expected-pre-inputs node)
                       (append (remove-instance symbol
                                                (expected-pre-inputs node))
                               (list (list symbol
                                           (eval (clause-function clause))))))
                 (return T)))))

;;; If 'output' is the first element of the clause body, then the cdr of the
;;; clause body is a function that will ultimately return the output of the
;;; goal (the goal result).
```

```
(defun integrate-output (clause)
  (let ((node (lookup-node (clause-name clause))))
    (cond (node
            (setf (output-func-args node) (arguments (clause-function clause)))
            (setf (output-function node) (eval (clause-function clause))))
           (T (format t "~A~%Not defined as a goal.  Ignoring it !!~%" clause))
           )))

;;; Returns the list of arguments of a lambda function.

(defun arguments (some-function)
  (cadadr some-function))

;;; This function creates a new node or modifies an existing one if we are
;;; reconsulting clauses.

(defun integrate-node (type clause)
  (let ((node (lookup-node (clause-name clause))))
    (cond ((null node) (create-node type clause))
           ((redefine-mode) (modify-node node type clause))
           (T (format t "~A~%Clause allready exists.  Ignoring it !!~%" clause))
           )))

;;; Returns T if the current consult mode is 'redefine'.

(defun redefine-mode ()
  (if (equal *consult-mode* *redefine*) T))

;;; Creates a new goal node in the goal base and initializes it with the
;;; given information in the clause.

(defun create-node (type clause)
  (let ((node (make-instance type)))
    (setf (goal node) clause)
    (setf (goal-args node) (clause-args clause))
    (modify-body node type clause)
    (add-node-to-goal-base (clause-name clause) node)
    ))

;;; Returns the arguments to a goal as given in the clause.

(defun clause-args (clause)
  (cadar clause))

;;; Adds the goal and its node with information to the goal base.

(defun add-node-to-goal-base (name node)
  (setf *goal-base* (cons (list name node) *goal-base*)))

;;; Adds a procedure or subgoals to the node depending on the caluse type.

(defun modify-body (node type clause)
  (if (equal type 'program)
      (setf (procedure node) (eval (body clause)))
      (setf (subgoals  node) (clause-subgoals clause))))
```

```
;;; Returns the body of the clause.

(defun body (clause)
  (cadr clause))

;;; Returns the subgoals in the body of the clause.

(defun clause-subgoals (clause)
  (cdadr clause))

;;; If the consult mode is 'redefine', then the node information can be
;;; altered without removing the goal from the goal base first.

(defun modify-node (node type clause)
    (setf (goal node) clause)
    (setf (goal-args node) (clause-args clause))
    (setf (expected-pre-inputs node) nil)
    (setf (expected-post-inputs node) nil)
    (setf (total-time node) 0)
    (setf (successes node) 1)
    (setf (failures node) 1)
    (setf (output-func-args node) nil)
    (setf (output-function node) nil)
    (modify-body node type clause))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File    : achieve.l
; Author  : Nils Thune
; Created : November 28, 1986
; Mode    : Common Lisp
;
; Purpose : Defines the methods, functions that can achieve a goal(s) in CAOS.
;           - achieve
;
; Copyright (c) 1986, Nils Thune.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;; This macro computes the run time of a goal and then updates the goal
;;; information in the node (total-time and if the goal failed or not).
;;; Since the macro is called inside methods only, I had to use self
;;; as an argument name refering to the object the method was called with.
;;; (if I used ex. node instead of self, node would be unbound)

(defmacro time-goal (func self)
  '(let (result
         run-time-2
         (run-time-1 (get-internal-run-time)))
     (PROGN (setf result ,func)
            (setf run-time-2 (get-internal-run-time))
            (cond (result
                   (setf (successes self) (+ 1 (successes self)))
                   (setf (total-time self)
                         (+ (total-time self)
                            (/ (- run-time-2 run-time-1)
                               internal-time-units-per-second))))
                  (T
                   (setf (failures self) (+ 1 (failures self)))))
            result)))

;;; Achieves the specified goal, if possible.

(defun achieve (goal)
  (let ((node (lookup-node* (car goal))))
    (if node (achieve-node node (pre-inputs goal node))
             (format t "~%~A is not defined as a goal.~%" goal)
      )))

;;; Does the same as lookup-node, but warns the user if the goal is not
;;; defined (if the goal is not defined, it is the same as we did not
;;; achieve the goal -> returns nil as result.  This might result in
;;; failure when least expected by the user.)

(defun lookup-node* (name)
  (cond ((assoc name *goal-base*) (cadr (assoc name *goal-base*)))
        (T (format t "WARNING: ~A is not defined as a goal.~%" name)
           (format t "Result of it defaults to nil.~%"))))

;;; Returns an alist with input variable name and associated values.

(defun pre-inputs (goal node)
```

```
  (mapcar #'list (goal-args node) (cadr goal)))

;;; OR-NODE: This achieve method must make sure that all pre-inputs are ok,
;;; then it must achieve one of the subgoals of the goal node and finally
;;; it must do the output of the goal node.  It uses previous experience to
;;; decide which path to traverse (in form of average run time and freq. of
;;; success.

(define-method (or-node achieve-node) (inputs)
  (if *trace* (format t "Activating: ~A~%" (goal self)))
  (let ((untried-subgoals (subgoals self)) (result nil))
    (time-goal
     (if (and
          (check-inputs (expected-pre-inputs self) inputs)
          (do ((best-subgoal (find-best-subgoal untried-subgoals)
                             (find-best-subgoal untried-subgoals)))
              ((null best-subgoal) failed)
              (if (setf result (achieve-node (node best-subgoal)
                                             (correct-inputs
                                              (args (goal (node best-subgoal)))
                                              (args (car best-subgoal))
                                              inputs)))
                (return (setf inputs (append inputs (list (list 's1 result)))))
                (setf untried-subgoals (remove-instance (caar best-subgoal)
                                                        untried-subgoals)))))
         (produce-output self inputs))
     self)))

;;; Makes sure that the functions in the alist of arguments and functions,
;;; returns T when applied to the matching argument value in the alist
;;; of arguments and values (inputs).

(defun check-inputs (functions inputs)
  (dolist (current functions T)
    (if (apply (cadr current) (cdr (assoc (car current) inputs)))
        T
        (return nil))))

;;; This function uses the previous experience stored in a goal to determine
;;; which path to follow (only called by an OR node controller).

(defun find-best-subgoal (subgoals)
  (let ((best-subgoal (lookup-goal (car subgoals))) (current-node nil))
    (dolist (current (cdr subgoals) best-subgoal)
      (setf best-subgoal (best-goal best-subgoal (lookup-goal current))))))

;;; Returns the best of two goals depending on freq. of success and path
;;; length. E(s) = Time/P(s) , where the smallest E(s) is chosen as the
;;; best goal (see write up for more details).

(defun best-goal (info1 info2)
  (let ((node1 (cadr info1)) (node2 (cadr info2)))
  (cond ((null node1) info2)
        ((null node2) info1)
        ((< (/ (* (total-time node1)
                  (+ (failures node1) (successes node1)))
```

```
                (* (successes node1) (successes node1)))
             (/ (* (total-time node2)
                   (+ (failures node2) (successes node2)))
                (* (successes node2) (successes node2))))
          info1)
         (T info2))))

;;; Returns a list (goal node-object) from the goal base.  Warns the user
;;; if a goal is not defined.

(defun lookup-goal (subgoal)
  (cond ((assoc (car subgoal) *goal-base*)
          (list subgoal (cadr (assoc (car subgoal) *goal-base*))))
         (T (format t "WARNING: ~A is not defined as a goal.~%" (car subgoal))
            (format t "Result of it defaults to nil.~%"))))

;;; Returns an alist of the correct arguments and their value for the
;;; goal to be achieved.

(defun correct-inputs (args1 args2 inputs)
  (cond ((null args2) nil)
         (T (cons (list (car args1) (cadr (assoc (car args2) inputs)))
                  (correct-inputs (cdr args1) (cdr args2) inputs)))))

;;; Returns the input arguments of a goal.

(defun args (goal)
  (cadr goal))

;;; Produces the output of a goal.  If no output function is provided for
;;; the goal the last result of the input list is returned (this would be
;;; the result of the last subgoal in the case of an AND-NODE, or the
;;; only subgoal result in the case of an OR-NODE.

(defun produce-output (node inputs)
  (if (check-inputs (expected-post-inputs node) inputs)
      (if (output-function node)
          (apply (output-function node)
                 (output-args (output-func-args node) inputs))
          (return-last-input inputs))
      failed))

;;; Returns the correct inputs needed for the output function.

(defun output-args (args inputs)
  (if args
      (cons (cadr (assoc (car args) inputs)) (output-args (cdr args) inputs))))

;;; Returns the value of the last input of the input list.

(defun return-last-input (inputs)
  (cadr (nth (- (length inputs) 1) inputs)))

;;; AND-NODE: This achieve method must make sure that all pre-inputs are ok,
;;; then it must achieve each of the subgoals of the goal node and finally
;;; it must do the output function of the goal node.
```

```
(define-method (and-node achieve-node) (inputs)
  (if *trace* (format t "Activating: ~A~%" (goal self)))
  (let ((n 0) (node nil) (result nil))
    (time-goal
     (if (and
           (check-inputs (expected-pre-inputs self) inputs)
           (dolist (subgoal (subgoals self) inputs)
             (setf n (+ n 1))
             (if (and
                   (setf node (lookup-node* (car subgoal)))
                   (setf result (achieve-node
                                  node
                                  (correct-inputs (goal-args node)
                                                  (args subgoal) inputs))))
                 (setf inputs (append inputs (list (list (sm n) result))))
                 (return failed))))
         (produce-output self inputs))
     self)))

;;; Returns a symbol of the form sm (s1 or s2, etc.) for use in the list
;;; of inputs (pre and post inputs) to the goal (sm is a post input).

(defun sm (n)
  (cadr (assoc n *subgoal-variables*)))

;;; PROGRAM: This achieve method must make sure that all pre-inputs are ok,
;;; then it must achieve the goal node by executing the procedure of the goal
;;; node and make sure that the post-inputs from it are ok, and finally it
;;; must do the output function of the goal node.

(define-method (program achieve-node) (inputs)
  (if *trace* (format t "Activating: ~A~%" (goal self)))
  (time-goal
   (if (check-inputs (expected-pre-inputs self) inputs)
       (produce-output
        self
        (append inputs (list (list 's1 (apply (procedure self)
                                              (input-values inputs))))))
       failed)
   self))

;;; Returns the values of the arguments found in the input alist.

(defun input-values (inputs)
  (mapcar #'cadr inputs))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File    : tools.l
; Author  : Nils Thune
; Created : November 25, 1986
; Mode    : Common Lisp
;
; Purpose : Defines some tools for CAOS.
;           - display
;           - erase
;           - trace-on
;           - trace-off
;
; Copyright (c) 1986, Nils Thune.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;; Displays information about all (or one particular) goal(s) in the
;;; knowledge base.

(defun display (&optional goal)
  (cond ((null goal) (dolist (current *goal-base*)
                              (print-information (node current))))
        ((lookup-node goal) (print-information (lookup-node goal)))
        (T (format t "~A is not defined as a goal.~%" goal))
        ))

;;; Prints out information from the slots of the goal (node) object.

(defun print-information (node)
  (format t "~%Goal: ~A~%" (goal node))
  (format t "Goal args: ~A~%" (goal-args node))
  (format t "Expected pre inputs: ~A~%" (expected-pre-inputs node))
  (format t "Expected post inputs: ~A~%" (expected-post-inputs node))
  (format t "Average run time: ~8,2f seconds~%" (/ (total-time node)
                                                  (successes node)))
  (format t "P(S) = ~1,1f~%" (/ (successes node)
                               (+ (successes node) (failures node))))
  (format t "Output args: ~A~%" (output-func-args node))
  (format t "Output function : ~A~%" (output-function node))
  )

;;; Erases the entire knowlegde base, or only one goal if specified.

(defun erase (&optional goal)
  (if goal (setf *goal-base* (remove-instance goal *goal-base*))
          (setf *goal-base* nil))
  (format t "Erased OK !!~%"))

;;; Removes one goal from the knowledge base.

(defun remove-instance (element assoc-list)
  (cond ((null assoc-list) nil)
        ((equal element (caar assoc-list)) (cdr assoc-list))
        (T (append (list (car assoc-list))
                   (remove-instance element (cdr assoc-list))))))
```

```
;;; Returns the node object associated with a goal.

(defun node (goal-list)
  (cadr goal-list))

;;; Sets the trace mode to ON. (Initialized to nil at startup)

(defun trace-on ()
  (setf *trace* T))

;;; Sets the trace mode to OFF.

(defun trace-off ()
  (setf *trace* nil))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File     : help.l
; Author   : Nils Thune
; Created  : November 25, 1986
; Mode     : Common Lisp
;
; Purpose  : Defines help functions for CAOS.
;            - help
;
; Copyright (c) 1986, Nils Thune.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;; Prints out all available commands in CAOS or information about a specific
;;; command.

(defun help (&optional command)
  (if command (help-command command) (display-help)))

(defun display-help ()
  (format t "
 Bellow is a listing of the currently available commands in CAOS.  For
 more specific help on any of them do a (help command).

 Command                Comments
 -------------------    -------------------------------------------------------
 (achieve goal)         Ahcieve specified goal.
 (consult clauses)      Add goals to knowledge base.
 (display)              Display all goals in the knowledge base.
 (display goal)         Display this particular goal.
 (erase)                Erase the knowledge base.
 (erase goal)           Erase goal from knowledge base.
 (help)                 Display all commands available.
 (help command)         Display help on this command.
 (reconsult clauses)    Redefine and add goals to knowledge base.
 (trace-on)             Set trace mode to on.
 (trace-off)            Set trace mode to off.
 "))

;;; Prints out the help in formation found under the command in the help
;;; list.  If no help is found, nil is printed.

(defun help-command (command)
  (format t "Current Available Help:~%~A" (cadr (assoc command *help-list*))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File    : graph-rout.l
; Author  : Nils Thune
; Created : November 17, 1986
;
; Copyright (c) 1986, Nils Thune.
;
; Purpose:
;
; To graphically show some concepts involved with CAOS:
;
; To display a scene, as seen through the users eyes from a specified
; viewpoint, in one window, and to display the same scene, as seen trough
; the camera of a robot moving in this scene, in a second window.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;;; Makes starbase graphics routines availbale.

(require "hp-ux_3g")

;;; This function sets the different parameters allowing the user to select
;;; how to view the scene as seen through the camera mounted on the robot.
;;; Note that the camera is fixed on the robot.  The function returns the
;;; correct transformation matrix.

(defun get-transformation (xp yp d)
  (set-view-reference-point 0.0 0.0 0.0)
  (set-view-plane-normal xp yp -1.0)
  (set-view-distance d)
  (make-view-plane-transformation))

;;; This function sets the different parameters allowing the user to select
;;; how to view the scene as seen through his own eyes.  The function returns
;;; the correct transformation matrix.

(defun get-user-transformation ()
  (set-view-reference-point 0.0 0.0 0.0)
  (set-view-plane-normal -0.15 -0.3 -1.0)
  (set-view-distance -5.0)
  (make-view-plane-transformation))

;;; Returns the perspective transformation matrix.

(defun get-perspective-transformation ()
  *perspective-transformation*)

;;; Returns the scaling matrix fro the camera.

(defun get-camera-scale ()
  *camera-scale*)

;;; Pops the TOS of the graphics stack (the current transformation matrix).
```

```
(defun pop-transformation-matrix ()
  (hp-ux_3g:pop_matrix *fd*)
  T)

;;; Switches to the window where the scene, as seen by the camera, is shown.

(defun current-window-camera ()
  (hp-ux_3g:set_p1_p2 *fd* hp-ux_3g:FRACTIONAL 0.72 0.72 0.0 1.0 1.0 1.0)
  T)

;;; Switches to the window where the scene, as seen by the user, is shown.

(defun current-window-scene ()
  (hp-ux_3g:set_p1_p2 *fd* hp-ux_3g:FRACTIONAL 0.0 0.3 0.0 0.7 1.0 1.0)
  T)

;;; This function sets up the windows for both the scene and camera.

(defun set-up-graphics-device ()
  (setf *fd* (hp-ux_3g:gopen device hp-ux_3g:outdev driver *open-mode*))
  (hp-ux_3g:flush_matrices *fd*)
  (hp-ux_3g:depth_indicator *fd* 1 1)
  (hp-ux_3g:vdc_extent *fd* WX1 WY1 WZ1 WX2 WY2 WZ2)
  (hp-ux_3g:mapping_mode *fd* 1)
  (hp-ux_3g:clip_rectangle *fd* WX1 WX2 WY1 WY2)
  (hp-ux_3g:clip_depth *fd* WZ2 WZ1)
  (hp-ux_3g:background_color *fd* 1.0 1.0 1.0)
  (current-window-scene)
  (gclear)
  (hp-ux_3g:background_color *fd* 1.0 0.8 0.9)
  (current-window-camera)
  (gclear)
  T)

;;; Clears the window of the specified file descriptor *fd*))

(defun gclear ()
  (hp-ux_3g:clear_control *fd* hp-ux_3g:clear_vdc_extent)
  (hp-ux_3g:clear_view_surface *fd*)
  (hp-ux_3g:make_picture_current *fd*)
  T)

;;; Closes the window(s) of the specified file descriptor.

(defun gend ()
  (hp-ux_3g:gclose *fd*)
  T)

;;; This function concatinates the given matrix with the current transformation
;;; matrix on the stack, and then pushes the result onto the stack.

(defun set-transformation-to (matrix)
  (hp-ux_3g:concat_transformation3d *fd* matrix hp-ux_3g:PRE hp-ux_3g:PUSH)
  T)
```

```
;;; For changing the view reference point.
;;; Arguments x,y,z the new view reference point.
;;; Global *xr*,*xy*,*xz* permanent storage for the reference point.

(defun set-view-reference-point (x y z)
  (setf *xr* x)
  (setf *yr* y)
  (setf *zr* z))

;;; For changing the view plane normal.
;;; Arguments dx,dy,dz the new view plane normal vector.
;;; Global *dxn*,*dyn*,*dzn* permanent storage for the view plane normal.

(defun set-view-plane-normal (dx dy dz)
  (let ((d (sqrt (+ (* dx dx) (* dy dy) (* dz dz)))))
    (setf *dxn* (/ dx d))
    (setf *dyn* (/ dy d))
    (setf *dzn* (/ dz d))))

;;; For changing the distance between the view reference point and the view
;;; plane.
;;; Argument d the new distance.
;;; Global *view-distance* the permanent storage for the view distance.

(defun set-view-distance (d)
  (setf *view-distance* d))

;;; For changing the direction that will be vertical on the image plane.
;;; Arguments dx,dy,dz the new view up vector.
;;; Global *dxup*,*dyup*,*dzup* permanent storage for view-up direction.

(defun set-view-up (dx dy dz)
  (setf *dxup* dx)
  (setf *dyup* dy)
  (setf *dzup* dz))

;;; For making the view-plane transformation.

(defun make-view-plane-transformation ()
  (let ((tmatrix 0) (v 0) (xup-vp 0) (yup-vp 0) (rup 0))
; Translate so that view plane center is new origin.
    (setf tmatrix (translate (- (+ *xr* (* *dxn* *view-distance*)))
                             (- (+ *yr* (* *dyn* *view-distance*)))
                             (- (+ *zr* (* *dzn* *view-distance*)))))
; Rotate so that view plane normal is Z axis.
    (setf v (sqrt (+ (* *dyn* *dyn*) (* *dzn* *dzn*))))
    (setf tmatrix (rotate 'x (- (/ *dyn* v)) (- (/ *dzn* v)) tmatrix))
    (setf tmatrix (rotate 'y *dxn* v tmatrix))
; Determine the view-up direction in these new coordinates.
    (setf xup-vp (+ (* *dxup* (aref tmatrix 0 0))
                    (* *dyup* (aref tmatrix 1 0))
                    (* *dzup* (aref tmatrix 2 0))))
    (setf yup-vp (+ (* *dxup* (aref tmatrix 0 1))
                    (* *dyup* (aref tmatrix 1 1))
                    (* *dzup* (aref tmatrix 2 1))))
; Determine rotation needed to make view-up vertical.
```

```
    (setf rup (sqrt (+ (* xup-vp xup-vp) (* yup-vp yup-vp))))
    (setf tmatrix (rotate 'z (/ xup-vp rup) (/ yup-vp rup) tmatrix))
    tmatrix))

;;; Rotation about an axis.
;;; Arguments matrix to add rotation to
;;;           axis to rotate around
;;;           sine, cosine the sine and cosine of the rotation angle.

(defun rotate (axis sine cosine &optional matrix)
  (let ((mat (make-identity 4)))
    (case axis
          ('z
           (setf (aref mat 0 0) cosine)
           (setf (aref mat 0 1) sine)
           (setf (aref mat 1 0) (- sine))
           (setf (aref mat 1 1) cosine))
          ('y
           (setf (aref mat 0 0) cosine)
           (setf (aref mat 0 2) (- sine))
           (setf (aref mat 2 0) sine)
           (setf (aref mat 2 2) cosine))
          ('x
           (setf (aref mat 1 1) cosine)
           (setf (aref mat 1 2) sine)
           (setf (aref mat 2 1) (- sine))
           (setf (aref mat 2 2) cosine))
          )
    (if matrix (multiply mat matrix) mat)))

;;; Multiplies two matrices and returns the result as a matrix.
;;; Arguments m1,m2 two matrices

(defun multiply (matrix1 matrix2)
  (let ((matrix3 nil)
        (dim1 (array-dimensions matrix1))
        (dim2 (array-dimensions matrix2)))
    (cond ((= (cadr dim1) (car dim2))
           (setf matrix3 (make-new-array (car dim1) (cadr dim2) 0.0))
           (dotimes (i (car dim1))
             (dotimes (j (cadr dim2))
               (dotimes (k (car dim2))
                 (setf (aref matrix3 i j) (+ (aref matrix3 i j)
                                             (* (aref matrix1 i k)
                                                (aref matrix2 k j)))))))
           matrix3)
          (T (error "Illegal to multiply ~A and ~A~%" matrix1 matrix2)))))

;;; Returns a matrix that accomplishes the given transformation.
;;; Arguments matrix to multiply
;;;           tx,ty,tz the amount of translation

(defun translate (tx ty tz &optional matrix)
  (let ((mat (make-identity 4)))
    (setf (aref mat 3 0) (float tx))
    (setf (aref mat 3 1) (float ty))
```

```
    (setf (aref mat 3 2) (float tz))
    (if matrix (multiply mat matrix) mat)))

;;; Creates and returns a nxn identity matrix.
;;; Argument n the matrix size.

(defun make-identity (n)
  (let ((matrix (make-new-array n n 0.0)))
    (dotimes (i n) (setf (aref matrix i i) 1.0))
    matrix))

;;; Create an array/matrix of float, size nxm.
;;; Arguments n,m,val is rows, columns and initial element value.

(defun make-new-array (n m val)
  (make-array `(,n ,m) :initial-element (float val) :element-type 'float))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File     : scene.l
; Author   : Nils Thune
; Created  : November 17, 1986
;
; Copyright  (c) 1986, Nils Thune.
;
; Purpose :
;
; This file defines six cubes (of different sizes and colors) used to draw
; a scene in a room.  The reason for using 6 (5*4) arrays, and initialize them
; to their corresponding cube corner values, is that the scene are redrawn
; several times showing different views of it.  If we had to set the points
; each time we drew the scene, it would slow down the program considerably.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Red cube

(setf p1 '(0.0 0.0 0.0))      ; Cube corners.
(setf p2 '(0.0 1.0 0.0))
(setf p3 '(0.0 0.0 -1.0))
(setf p4 '(0.0 1.0 -1.0))
(setf p5 '(1.0 0.0 -1.0))
(setf p6 '(1.0 1.0 -1.0))
(setf p7 '(1.0 0.0 0.0))
(setf p8 '(1.0 1.0 0.0))
                              ; cube ploygons
(setf sr1 (make-array '(4 3) :initial-contents (list p1 p2 p4 p3)))
(setf sr2 (make-array '(4 3) :initial-contents (list p3 p4 p6 p5)))
(setf sr3 (make-array '(4 3) :initial-contents (list p5 p6 p8 p7)))
(setf sr4 (make-array '(4 3) :initial-contents (list p7 p8 p2 p1)))
(setf sr5 (make-array '(4 3) :initial-contents (list p2 p4 p6 p8)))

;;; Blue cube.

(setf p1 '(2.0 0.0 -2.0))     ; cube corners.
(setf p2 '(2.0 3.0 -2.0))
(setf p3 '(2.0 0.0 -3.0))
(setf p4 '(2.0 3.0 -3.0))
(setf p5 '(3.0 0.0 -3.0))
(setf p6 '(3.0 3.0 -3.0))
(setf p7 '(3.0 0.0 -2.0))
(setf p8 '(3.0 3.0 -2.0))
                              ; cube ploygons
(setf sb1 (make-array '(4 3) :initial-contents (list p1 p2 p4 p3)))
(setf sb2 (make-array '(4 3) :initial-contents (list p3 p4 p6 p5)))
(setf sb3 (make-array '(4 3) :initial-contents (list p5 p6 p8 p7)))
(setf sb4 (make-array '(4 3) :initial-contents (list p7 p8 p2 p1)))
(setf sb5 (make-array '(4 3) :initial-contents (list p2 p4 p6 p8)))

;;; Green cube.

(setf p1 '(7.0 0.0 -1.0))     ; cube corners.
(setf p2 '(7.0 1.0 -1.0))
(setf p3 '(7.0 0.0 -3.0))
```

```
(setf p4 '(7.0 1.0 -3.0))
(setf p5 '(9.0 0.0 -3.0))
(setf p6 '(9.0 1.0 -3.0))
(setf p7 '(9.0 0.0 -1.0))
(setf p8 '(9.0 1.0 -1.0))
                                        ; cube ploygons
(setf sg1 (make-array '(4 3) :initial-contents (list p1 p2 p4 p3)))
(setf sg2 (make-array '(4 3) :initial-contents (list p3 p4 p6 p5)))
(setf sg3 (make-array '(4 3) :initial-contents (list p5 p6 p8 p7)))
(setf sg4 (make-array '(4 3) :initial-contents (list p7 p8 p2 p1)))
(setf sg5 (make-array '(4 3) :initial-contents (list p2 p4 p6 p8)))

;;; Yellow cube.

(setf p1 '(2.0 0.0 -15.0))               ; cube corners.
(setf p2 '(2.0 1.0 -15.0))
(setf p3 '(2.0 0.0 -18.0))
(setf p4 '(2.0 1.0 -18.0))
(setf p5 '(4.0 0.0 -18.0))
(setf p6 '(4.0 1.0 -18.0))
(setf p7 '(4.0 0.0 -15.0))
(setf p8 '(4.0 1.0 -15.0))
                                        ; cube ploygons
(setf sy1 (make-array '(4 3) :initial-contents (list p1 p2 p4 p3)))
(setf sy2 (make-array '(4 3) :initial-contents (list p3 p4 p6 p5)))
(setf sy3 (make-array '(4 3) :initial-contents (list p5 p6 p8 p7)))
(setf sy4 (make-array '(4 3) :initial-contents (list p7 p8 p2 p1)))
(setf sy5 (make-array '(4 3) :initial-contents (list p2 p4 p6 p8)))

;;; AA cube.

(setf p1 '(3.0 0.0 -8.0))               ; cube corners.
(setf p2 '(3.0 1.0 -8.0))
(setf p3 '(3.0 0.0 -11.0))
(setf p4 '(3.0 1.0 -11.0))
(setf p5 '(7.0 0.0 -11.0))
(setf p6 '(7.0 1.0 -11.0))
(setf p7 '(7.0 0.0 -8.0))
(setf p8 '(7.0 1.0 -8.0))
                                        ; cube ploygons
(setf sa1 (make-array '(4 3) :initial-contents (list p1 p2 p4 p3)))
(setf sa2 (make-array '(4 3) :initial-contents (list p3 p4 p6 p5)))
(setf sa3 (make-array '(4 3) :initial-contents (list p5 p6 p8 p7)))
(setf sa4 (make-array '(4 3) :initial-contents (list p7 p8 p2 p1)))
(setf sa5 (make-array '(4 3) :initial-contents (list p2 p4 p6 p8)))

;;; CC cube.

(setf p1 '(6.0 0.0 -14.0))               ; cube corners.
(setf p2 '(6.0 2.0 -14.0))
(setf p3 '(6.0 0.0 -16.0))
(setf p4 '(6.0 2.0 -16.0))
(setf p5 '(7.0 0.0 -16.0))
(setf p6 '(7.0 2.0 -16.0))
(setf p7 '(7.0 0.0 -14.0))
(setf p8 '(7.0 2.0 -14.0))
```

```
                                        ; cube ploygons
(setf sc1 (make-array '(4 3) :initial-contents (list p1 p2 p4 p3)))
(setf sc2 (make-array '(4 3) :initial-contents (list p3 p4 p6 p5)))
(setf sc3 (make-array '(4 3) :initial-contents (list p5 p6 p8 p7)))
(setf sc4 (make-array '(4 3) :initial-contents (list p7 p8 p2 p1)))
(setf sc5 (make-array '(4 3) :initial-contents (list p2 p4 p6 p8)))

(setf p1 '(0.0 0.0 0.0))      ; wall corners of room.
(setf p2 '(0.0 10.0 0.0))
(setf p3 '(0.0 10.0 -20.0))
(setf p4 '(0.0 0.0 -20.0))
                                        ; wall ploygon
(setf left (make-array '(4 3) :initial-contents (list p1 p2 p3 p4)))

(setf p1 '(0.0 0.0 -20.0))      ; wall corners.
(setf p2 '(0.0 10.0 -20.0))
(setf p3 '(10.0 10.0 -20.0))
(setf p4 '(10.0 0.0 -20.0))
                                        ; wall ploygon
(setf back (make-array '(4 3) :initial-contents (list p1 p2 p3 p4)))

(setf p1 '(10.0 0.0 0.0))      ; wall corners.
(setf p2 '(10.0 10.0 0.0))
(setf p3 '(10.0 10.0 -20.0))
(setf p4 '(10.0 0.0 -20.0))
                                        ; wall ploygon
(setf right (make-array '(4 3) :initial-contents (list p1 p2 p3 p4)))

(setf p1 '(0.0 0.0 0.0))      ; wall corners.
(setf p2 '(0.0 0.0 -20.0))
(setf p3 '(10.0 0.0 -20.0))
(setf p4 '(10.0 0.0 0.0))
                                        ; wall ploygon
(setf floor (make-array '(4 3) :initial-contents (list p1 p2 p3 p4)))

;;; Draws the scene, which contains several cubes of different sizes
;;; and colors.

(defun draw-scene ()
  (hp-ux_3g:fill_color *fd* 1.0 0.8 0.6)
  (hp-ux_3g:perimeter_color *fd* 0.0 0.0 0.0)
  (hp-ux_3g:interior_style *fd* hp-ux_3g:INT_SOLID 1)
  (hp-ux_3g:polygon3d *fd* floor 4 0)
  (hp-ux_3g:polygon3d *fd* left 4 0)
  (hp-ux_3g:polygon3d *fd* back 4 0)
  (hp-ux_3g:polygon3d *fd* right 4 0)
  (draw-cube sy1 sy2 sy3 sy4 sy5 0.0 1.0 1.0)
  (draw-cube sc1 sc2 sc3 sc4 sc5 1.0 1.0 0.0)
  (draw-cube sa1 sa2 sa3 sa4 sa5 1.0 0.0 1.0)
  (draw-cube sr1 sr2 sr3 sr4 sr5 1.0 0.0 0.0)
  (draw-cube sb1 sb2 sb3 sb4 sb5 0.0 0.0 1.0)
  (draw-cube sg1 sg2 sg3 sg4 sg5 0.0 1.0 0.0)
  (hp-ux_3g:make_picture_current *fd*)
  T)

;;; Draws a cube with 4 sides and a top.  The sides and the top gets the
```

```
;;; color "r g b".

(defun draw-cube (s1 s2 s3 s4 s5 r g b)
    (hp-ux_3g:fill_color *fd* r g b)
    (hp-ux_3g:perimeter_color *fd* 0.0 0.0 0.0)
    (hp-ux_3g:interior_style *fd* hp-ux_3g:INT_SOLID 1)
    (hp-ux_3g:polygon3d *fd* s1 4 0)
    (hp-ux_3g:polygon3d *fd* s2 4 0)
    (hp-ux_3g:polygon3d *fd* s3 4 0)
    (hp-ux_3g:polygon3d *fd* s4 4 0)
    (hp-ux_3g:polygon3d *fd* s5 4 0)
    T)
```

8. Appendix B

CAOS Source Code: C Version

This appendix lists the complete C source code of CAOS.

```
/****************************************************************************

Procedure Name   : Achieve_Goal()
Part of          : Hierarchical Robot Control System
File Name        : ag.c
Date             : 05.15.86
Author           : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose          : To achieve a goal (this proceudre is the schema itself)

****************************************************************************/

#include         "defs.h"        /* user defined goodies */

void Achieve_Goal(Goal,Goal_Information)
char             Goal[];
goalinfo         *Goal_Information;
{
boolean          status = GO;

printf("Neuroschema activated\n      for : %s\n",Goal);

while (status != DONE && status != QUIT)
{

/*___________________________________________________________________________
 * Activation Section
 */

        status = Check_Goal_Status(Goal_Information);


/*___________________________________________________________________________
 * Event Section
 */

        if (status == GO) status = Achieve_SubGoals(Goal_Information);


/*___________________________________________________________________________
 * Learning Section
 */


        if (status == LEARN) status = Learn_From_User(Goal,Goal_Information);

/*___________________________________________________________________________
 */

}
Print_Goal_Result(Goal,Goal_Information);
Update_Probabilities(Goal,Goal_Information);
}
```

```c
/***************************************************************************

Procedure Name   : Assign_Inputs()
Part of          : Hierarchical Robot Control System
File Name        : ai.c
Date             : 05.15.86
Author           : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose          :

***************************************************************************/

#include         "defs.h"        /* user defined goodies */

void Assign_Inputs(Goal,Goal_Info)
char             Goal[];
goalinfo         *Goal_Info;
{
        int                 no;
        int                 level = 0;
        input               *Inp;
        char                Argument[50];

        Inp = Goal_Info->Needed_Input;

        while (Inp != NULL)
        {
                while (Goal[++level] != ARG) ;
                no = 0;
                while (Goal[++level] != ARG)
                {
                        if (Goal[level] == '\0') ERROR_MSG("ai.c: Missing ~ when
                        Argument[no++] = Goal[level];
                }
                Argument[no] = '\0';
                Inp->Value = MORE_MEMORY(1+no,char);
                strcpy(Inp->Value,Argument);
                Inp = Inp->Next;
        }
}
```

```
/*************************************************************************

Procedure Name    : Assign_Names()
Part of           : Hierarchical Robot Control System
File Name         : an.c
Date              : 05.15.86
Author            : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose           : To make names match both to level above and below


*************************************************************************/

#include        "defs.h"        /* user defined goodies */

void Assign_Names(Goal,Goal_Info)
char            Goal[];
goalinfo        *Goal_Info;
{
        int     i, no;
        input   *Inp;
        output  *Out;
        char    value[50];

        if (Goal_Info->Null) return;
        i = -1;
        Inp = Goal_Info->Needed_Input;

        while (Goal[++i] != '\0' && Goal[i] != EOC && Goal[i] != OUTP)
        if (Goal[i] == ARG)
        {
                no = 0;
                while (Goal[++i] != ARG) value[no++] = Goal[i];
                Inp->Inp_Name1 = MORE_MEMORY(1+no,char);
                strncpy(Inp->Inp_Name1,value,no);
                Inp = Inp->Next;
        }
        Out = Goal_Info->Needed_Output;
        while (Goal[++i] != '\0' && Goal[i] != EOC)
        if (Goal[i] == ARG)
        {
                no = 0;
                while (Goal[++i] != ARG) value[no++] = Goal[i];
                Out->Out_Name1 = MORE_MEMORY(1+no,char);
                strncpy(Out->Out_Name1,value,no);
                Out = Out->Next;
        }
}
```

```
/**********************************************************************

Procedure Name  : Achieve_SubGoals()
Part of         : Hierarchical Robot Control System
File Name       : as.c
Date            : 05.15.86
Author          : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose         : To achieve the subgoals of a goal

**********************************************************************/

#include        "defs.h"        /* user defined goodies */

int Achieve_SubGoals(Goal_Information)
goalinfo        *Goal_Information;
{
        subgoal *SubGoal;
        int     Sgr;

        SubGoal = Find_Best_AND_group(Goal_Information);
        if (SubGoal == NULL) return(QUIT);
        Sgr = SubGoal->Group;
        printf("Group = %d\n",Sgr);
        while (SubGoal != NULL && SubGoal->Group == Sgr)
        {
                SubGoal->Goal_Info = Get_Goal_Information(SubGoal->SubGoal);
                Assign_Names(SubGoal->SubGoal,SubGoal->Goal_Info);
                if (Get_Input(Goal_Information,SubGoal) == NOTOUTOK) return;

                if (SubGoal->Goal_Info->ProgNo > 0)
                {
                        Execute(SubGoal);
                }
                else
                {
                        Achieve_Goal(SubGoal->SubGoal,SubGoal->Goal_Info);
                }
                if (SubGoal->Goal_Info->OutOK == FALSE) return;
                SubGoal = SubGoal->Next;
        }
}
```

```
/***************************************************************************

Procedure Name  : Check_Goal_Status()
Part of         : Hierarchical Robot Control System
File Name       : cgs.c
Date            : 05.15.86
Author          : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose         : To see if the goal is done or if it must learn from the
                  user to achieve the goal

***************************************************************************/

#include        "defs.h"        /* user defined goodies */

boolean Check_Goal_Status(Goal_Information)
goalinfo        *Goal_Information;
{
        subgoal *SubGoal;
        int     Sgr;

        if (Goal_Information->Null) return(LEARN);

        SubGoal = Find_Best_AND_group(Goal_Information);
        if (SubGoal == NULL) return(QUIT);
        if (SubGoal->Goal_Info == NULL) return(GO);
        Sgr = SubGoal->Group;

        while (SubGoal != NULL && Sgr == SubGoal->Group)
        {
                if (!SubGoal->Goal_Info->OutOK)
                {
                        Remove_SubGoal(Goal_Information,Sgr);
                        if (Goal_Information->SubGoals == NULL) break;
                        else return(GO);
                }
                SubGoal = SubGoal->Next;
        }
        Get_Output(Goal_Information);
        return(DONE);
}
```

```
/************************************************************************

Procedure Name  : Execute()
Part of         : Hierarchical Robot Control System
File Name       : e.c
Date            : 05.15.86
Author          : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose         : To execute a program

************************************************************************/

#include        "defs.h"        /* user defined goodies */

void Execute(SubGoal)
subgoal *SubGoal;
{
        output  *Out;

/*      printf("Executing :
%d%s\n",SubGoal->Goal_Info->ProgNo,SubGoal->SubGoal); */
        switch(SubGoal->Goal_Info->ProgNo)
        {
                case    1:      prog1(SubGoal); break;

                case    2:      prog2(SubGoal); break;

                case    3:      prog3(SubGoal); break;

                case    4:      prog4(SubGoal); break;

                case    5:      prog5(SubGoal); break;

                case    6:      prog6(SubGoal); break;

                case    7:      prog7(SubGoal); break;

                default:        ERROR_MSG("Execute Error","");
        }
        Print_Goal_Result(SubGoal->SubGoal,SubGoal->Goal_Info);
        Update_Probabilities(SubGoal->SubGoal,SubGoal->Goal_Info);
}
```

```c
/*****************************************************************************

Procedure Name  : Find_Best_AND_group()
Part of         : Hierarchical Robot Control System
File Name       : fbag.c
Date            : 05.15.86
Author          : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose         : To find the best AND gropu for achieving a goal

*****************************************************************************/

#include        "defs.h"        /* user defined goodies */

subgoal *Find_Best_AND_group(Goal_Info)
goalinfo        *Goal_Info;
{
        subgoal *SubGoal;
        subgoal *Temp;
        int     Sgr;
        float   Prob;
        float   NewProb;

        SubGoal = Goal_Info->SubGoals;
        if (SubGoal == NULL) return(SubGoal);
        Temp = SubGoal;
        Sgr = SubGoal->Group;
        Prob = Find_Prob(SubGoal->SubGoal);

        for (;;)
        {
                while (Temp != NULL && Temp->Group == Sgr) Temp = Temp->Next;

                if (Temp == NULL) break;
                NewProb = Find_Prob(Temp->SubGoal);
                if (NewProb > Prob)
                {
                        Prob = NewProb;
                        SubGoal = Temp;
                }
                Sgr = Temp->Group;
        }
        return(SubGoal);
}
```

```
/*************************************************************************

Procedure Name   : Find_Prob()
Part of          : Hierarchical Robot Control System
File Name        : fp.c
Date             : 05.15.86
Author           : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose          : To find probability for a goal

*************************************************************************/

#include        "defs.h"                /* user defined goodies */

float Find_Prob(Goal)
char    Goal[];                         /* command string */
{
        extern Kelem    *Know_Table[27];/* know. base */
        Kelem           *Curr_Elem;

        Curr_Elem = Know_Table[Goal[0]-'A'];

        if (Curr_Elem == NULL) return(0.0);

        while (!GoalComp(Goal,Curr_Elem->Goal)) Curr_Elem = Curr_Elem->Next;

        return(Curr_Elem->S/(float)Curr_Elem->N);
}
```

```
/***************************************************************************

Procedure Name  : GoalComp()
Part of         : Hierarchical Robot Control System
File Name       : gc.c
Date            : 05.15.86
Author          : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose         : To compare to goals and see if they match

***************************************************************************/

#include        "defs.h"                /* user defined goodies */

boolean GoalComp(goal1,goal2)
char    *goal1;
char    *goal2;
{
        int     i = 0;
        int     j = 0;

        while (goal1[i] == goal2[j])
        {
                if (goal1[i] == ARG)
                {
                        while (goal1[++i] != ARG) ;
                        while (goal2[++j] != ARG) ;
                }
                ++i;
                ++j;
                if (goal1[i] == OUTP || goal1[i] == EOC || goal1[i] == '\0')
                        return(TRUE);
        }
        return(FALSE);
}
```

```
/**************************************************************************

Procedure Name   : Get_Goal_Information()
Part of          : Hierarchical Robot Control System
File Name        : ggi.c
Date             : 05.15.86
Author           : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose          : To get the goal information from the knowledge base for a
                   a goal


**************************************************************************/

#include        "defs.h"        /* user defined goodies */

goalinfo *Get_Goal_Information(Goal)
char    Goal[];
{
        extern Kelem        *Know_Table[27];
        Kelem               *Curr_Elem;
        goalinfo            *Goal_Info;
        subgoal             *Subgl, *Subg;
        input               *inp1, *inp2;
        output              *out1, *out2;
        char                Argument[50];

        Curr_Elem = Know_Table[Goal[0]-'A'];
        while (Curr_Elem != NULL)
        {
                if (GoalComp(Goal,Curr_Elem->Goal)) break;
                Curr_Elem = Curr_Elem->Next;
        }
        Goal_Info = MORE_MEMORY(1,goalinfo);
        if (Curr_Elem == NULL)                  /* goal does not exists */
        {
                Goal_Info->Null = TRUE;
                return(Goal_Info);
        }
        Goal_Info->ProgNo = Curr_Elem->Goal_Info->ProgNo;

        /* copy over input information */

        inp1 = Curr_Elem->Goal_Info->Needed_Input;
        if (inp1 != NULL)
        {
            inp2 = MORE_MEMORY(1,input);
            Goal_Info->Needed_Input = inp2;
            while (inp1 != NULL)
            {
                inp2->Inp_Name = inp1->Inp_Name;
                inp1 = inp1->Next;
                if (inp1 != NULL)
                {
                        inp2->Next = MORE_MEMORY(1,input);
                        inp2 = inp2->Next;
                }
            }
        }
        /* copy over output information */

        Goal_Info->Output = Curr_Elem->Goal_Info->Output;

        out1 = Curr_Elem->Goal_Info->Needed_Output;
```

```
        if (out1 != NULL)
        {
            out2 = MORE_MEMORY(1,output);
            Goal_Info->Needed_Output = out2;
            while (out1 != NULL)
            {
                out2->Out_Name = out1->Out_Name;
                out1 = out1->Next;
                if (out1 != NULL)
                {
                        out2->Next = MORE_MEMORY(1,output);
                        out2 = out2->Next;
                }
            }
        }
        /* copy over subgoal information */

        Subg1 = Curr_Elem->Goal_Info->SubGoals;
        if (Subg1 != NULL)
        {
            Goal_Info->SubGoals = MORE_MEMORY(1,subgoal);
            Subg = Goal_Info->SubGoals;
            while(Subg1 != NULL)
            {
                Subg->SubGoal = Subg1->SubGoal;
                Subg->Group = Subg1->Group;
                Subg1 = Subg1->Next;
                if (Subg1 != NULL)
                {
                        Subg->Next = MORE_MEMORY(1,subgoal);
                        Subg = Subg->Next;
                }
            }
        }
        return(Goal_Info);
}
```

```
/***************************************************************************

Procedure Name   : Get_Input()
Part of          : Hierarchical Robot Control System
File Name        : gi.c
Date             : 05.15.86
Author           : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose          : To get the input for a goal

***************************************************************************/

#include         "defs.h"        /* user defined goodies */

int Get_Input(Goal_Information,SubGoal)
goalinfo         *Goal_Information;
subgoal          *SubGoal;
{
        input   *Input1;
        input   *Input2;
        output  *Output;
        subgoal *Subg;
        boolean ALL = TRUE;
        boolean FOUND;

        if (SubGoal->Goal_Info->Null) return(1);

/* look for input values in main goal information first */

        Input2 = SubGoal->Goal_Info->Needed_Input;
        while (Input2 != NULL)
        {
                Input1 = Goal_Information->Needed_Input;
                while (Input1 != NULL)
                {
                        if (strcmp(Input1->Inp_Name,Input2->Inp_Name1)  == 0)
                        {
                                if (Input1->Value == NULL) ERROR_MSG("gi.c: Missin
                                Input2->Value = Input1->Value;
                                break;
                        }
                        Input1 = Input1->Next;
                }
                if (Input1 == NULL) ALL = FALSE;
                Input2 = Input2->Next;
        }
        if (ALL) return(1);

/* look for inputs among other subgoals outputs */

        Input1 = SubGoal->Goal_Info->Needed_Input;
        while (Input1 != NULL)
        {
                FOUND = FALSE;
                Subg = Goal_Information->SubGoals;
                if (Input1->Value == NULL)
                while (Subg != NULL)
                {
                    if (Subg == SubGoal)
                    {
                        User_Provide(SubGoal,Input1);
                        FOUND = TRUE;
                    }
                    else
```

```
                    {
                        Output = Subg->Goal_Info->Needed_Output;
                        while (Output != NULL)
                        {
                            if (strcmp(Output->Out_Name1,Input1->Inp_Name1)  == 0)
                            {
                                if (!Subg->Goal_Info->OutOK) return(NOTOUTOK);
                                Input1->Value = Output->Value;
                                FOUND = TRUE;
                                break;
                            }
                            Output = Output->Next;
                        }
                    }
                    if (FOUND) break;
                    Subg = Subg->Next;
                }
                if (Subg == NULL) User_Provide(SubGoal,Input1);
                Input1 = Input1->Next;
        }
        return(1);
}
```

```
/***************************************************************************

Procedure Name  : Get_Output()
Part of         : Hierarchical Robot Control System
File Name       : go.c
Date            : 05.15.86
Author          : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose         : To get the output for a goal

***************************************************************************/

#include        "defs.h"        /* user defined goodies */

void Get_Output(Goal_Information)
goalinfo        *Goal_Information;
{
        output  *Output1;
        output  *Output2;
        subgoal *Subg;
        boolean NotAll = FALSE;
        boolean FOUND;

        if (Goal_Information->SubGoals == NULL)
        {
                Goal_Information->Done = TRUE;
                Goal_Information->OutOK = FALSE;
                return;
        }
        Output1 = Goal_Information->Needed_Output;

        while (Output1 != NULL)
        {
            Subg = Goal_Information->SubGoals;
            while (Subg != NULL)
            {
                Output2 = Subg->Goal_Info->Needed_Output;
                while(Output2 != NULL)
                {
                    if (strcmp(Output1->Out_Name,Output2->Out_Name1) == 0)
                    {
                        Output1->Value = Output2->Value;
                        break;
                    }
                    Output2 = Output2->Next;
                }
                if (Output2 != NULL) break;
                Subg = Subg->Next;
            }
            if (Subg == NULL) ERROR_MSG("go.c: Can't find output !","");
            Output1 = Output1->Next;
        }
        Goal_Information->Done = TRUE;
        Goal_Information->OutOK = TRUE;
}
```

```
/***************************************************************************

Procedure Name   : Insert()
Part of          : Hierarchical Robot Control System
File Name        : i.c
Date             : 05.15.86
Author           : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose          :

***************************************************************************/

#include        "defs.h"                /* user defined goodies */

void Insert(com,Goal_Info,fd)
char            com[];                  /* command string */
goalinfo        *Goal_Info;             /* pointer to goal information */
FILE            *fd;
{
        extern Kelem    *Know_Table[27];/* know. base */
        Kelem           *Curr_Elem;

        Curr_Elem = MORE_MEMORY(1,Kelem);
        Curr_Elem->Goal = MORE_MEMORY(1+strlen(com),char);
        strcpy(Curr_Elem->Goal,com);

        Store_Arguments(com,Goal_Info);
        Read_In_Probs(fd,Curr_Elem);

        Curr_Elem->Goal_Info = Goal_Info;
        Curr_Elem->Next = Know_Table[com[0]-'A'];
        Know_Table[com[0]-'A'] = Curr_Elem;
}
```

```
/**********************************************************************

Procedure Name    : Learn_From_User()
Part of           : Hierarchical Robot Control System
File Name         : lfu.c
Date              : 05.15.86
Author            : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose           : To leanr from the user how to achieve the goal

**********************************************************************/

#include        "defs.h"        /* user defined goodies */

int Learn_From_User(Goal,Goal_Information)
char            Goal[];
goalinfo        *Goal_Information;
{
        goalinfo        *Goal_Info;
        subgoal         *Temp;
        subgoal         *Prev;
        int             i, Sgr;
        int             no;
        char            Goal1[COMMAND_SIZE];
        char            SubGoal[COMMAND_SIZE];

        if (Goal_Information->Null)
        {
                printf("\nI don't understand the meaning of \"%s\"\n\n",Goal);
                printf("Will you explain for me what to do ?(y/n): ");
                scanf("%s",SubGoal);
                switch(SubGoal[0])
                {
                        case 'y':
                        case 'Y':       break;
                        default:        printf("\nOK, Bye bye . . .\n\n");
                                        return(DONE);
                }

                Goal_Info = MORE_MEMORY(1,goalinfo);
                Temp = MORE_MEMORY(1,subgoal);
                Prev = Temp;
                Goal_Info->SubGoals = Temp;

                printf("Command: ");
                for (i = 2; i < COMMAND_SIZE; i++) printf("_");
                for (i = 2; i < COMMAND_SIZE; i++) printf("\b");
                fflush(stdout);
                scanf("%s",Goal1);
                TOUPPER(Goal1);
                Sgr = 1;
                no = 0;
                while(Sgr > 0)
                {
                        printf("\n0 = No More SubGoals\n");
                        printf("SubGoal Number %d ......\n",++no);
                        printf("AND Group # ? : ");
                        scanf("%d",&Sgr);
                        if (Sgr > 0)
                        {
                                printf("SubGoal         : ");
                                for (i = 8; i < COMMAND_SIZE; i++) printf("_");
                                for (i = 8; i < COMMAND_SIZE; i++) printf("\b");
                                fflush(stdout);
```

```
                                        scanf("%s",SubGoal);
                                        TOUPPER(SubGoal);

                                        Temp->SubGoal = MORE_MEMORY(1+strlen(SubGoal),cha
                                        strcpy(Temp->SubGoal,SubGoal);
                                        Temp->Group = Sgr;
                                        Prev = Temp;
                                        Temp->Next = MORE_MEMORY(1,subgoal);
                                        Temp = Temp->Next;
                                }
                        }
                        Prev->Next = NULL;
                        cfree(Temp);
                        Insert(Goal1,Goal_Info);
                        Goal_Info = Get_Goal_Information(Goal1);
                        if (Goal_Info->Null) {printf("\nNull info found: %s\n%s\n",Goal,G
                        Goal_Information->Needed_Input = Goal_Info->Needed_Input;
                        Goal_Information->Needed_Output = Goal_Info->Needed_Output;
                        Goal_Information->SubGoals = Goal_Info->SubGoals;
                        Goal_Information->Null = FALSE;
                        Assign_Inputs(Goal,Goal_Information);
                        return;
                }
                printf("Don't now how to learn this way yet !!!!!!!!\n\n");
                exit(1);
        }
```

```
/***************************************************************************

Procedure Name  : Load_Information()
Part of         : Hierarchical Robot Control System
File Name       : li.c
Date            : 05.15.86
Author          : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose         : To load the knowledge- and data base into memory

***************************************************************************/

#include        "defs.h"                        /* user defined goodies */

void Load_Information(KnowBase,ProgBase)
char    *KnowBase;                              /* file name for knowledge base *
char    *ProgBase;                              /* file name for program   base *
{
        FILE            *fd;                    /* file descriptor */
        FILE            *fopen();               /* open function   */
        extern  Kelem   *Know_Table[27];        /* know. base hash table */

        INITIALIZE(27,NULL,Know_Table);

        printf("\n\nTrying to Load Program Base . . . . ");
        if ((fd = fopen(ProgBase,"r")) == NULL) ERROR_MSG("Can't open ",ProgBase)
        Load_Program_Base(fd);                  /* load in program base */
        fclose(fd);                             /* close file */
        printf("Loaded !\n");

        printf("Trying to Load Knowledge Base . . . ");
        if ((fd = fopen(KnowBase,"r")) == NULL) ERROR_MSG("Can't open ",KnowBase)
        Load_Knowledge_Base(fd);                /* load in knowledge base */
        fclose(fd);                             /* close file */
        printf("Loaded !\n");
}
```

```
/***************************************************************************

Procedure Name   : Load_Knowledge_Base()
Part of          : Hierarchical Robot Control System
File Name        : lkb.c
Date             : 05.15.86
Author           : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose          : To load the knowledge base into memory

***************************************************************************/

#include        "defs.h"                /* user defined goodies */

void Load_Knowledge_Base(fd)
FILE    *fd;                            /* file descriptor for knowledge file */
{
        int     i;                      /* loop counter */
        char    com[COMMAND_SIZE];      /* character used to read file */
        char    subcom[COMMAND_SIZE];   /* character used to read file */
        int             no;             /* # of subgoals */
        int             Sno;            /* Subgoal group # */
        goalinfo        *Goal_Info;     /* pointer to goal information */
        subgoal         *Subg;          /* temporary pointer to subgoals */

        while (fscanf(fd,"%d%s!",&no,com) != EOF)
        {
                Goal_Info = MORE_MEMORY(1,goalinfo);
                Goal_Info->SubgNo = no;
                Subg = MORE_MEMORY(1,subgoal);
                Goal_Info->SubGoals = Subg;
                for (i = 0; i < no; ++i)
                {
                        fscanf(fd,"%d%s!",&Sno,subcom);
                        Subg->SubGoal = MORE_MEMORY(1+strlen(subcom),char);
                        strcpy(Subg->SubGoal,subcom);
                        Subg->Group = Sno;
                        if (i < no-1)
                        {
                                Subg->Next = MORE_MEMORY(1,subgoal);
                                Subg = Subg->Next;
                        }
                }
                Insert(com,Goal_Info,fd);
        }
}
```

```
/*****************************************************************************

Procedure Name  : Load_Program_Base()
Part of         : Hierarchical Robot Control System
File Name       : lpb.c
Date            : 05.15.86
Author          : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose         : To load in low level programs into the know. base

*****************************************************************************/

#include        "defs.h"                /* user defined goodies */

void Load_Program_Base(fd)
FILE    *fd;                            /* file descriptor for program base */
{
        goalinfo        *Goal_Info;     /* pointer to goalinfo */
        char    com[COMMAND_SIZE];      /* character used to read file */
        int             Pno;            /* program number */

        while (fscanf(fd,"%d%s!",&Pno,com) != EOF)
        {
                Goal_Info = MORE_MEMORY(1,goalinfo);
                Goal_Info->ProgNo = Pno;
                Insert(com,Goal_Info,fd);
        }
}
```

```
/*************************************************************************

Procedure Name   : main()
Part of          : Hierarchical Robot Control System
File Name        : main.c
Date             : 05.15.86
Author           : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose          : To control all the schemas and low level programs that
                   are needed to achieve a goal

*************************************************************************/

#include        "defs.h"                /* user defined goodies */

Kelem   *Know_Table[27];                /* Hash table of goals */

main()
{
        char    Main_Goal[COMMAND_SIZE];/* main goal from user */
        goalinfo *Goal_Information;     /* pointer to goal information */
        char    KnowBase[50];           /* file name for know. base */
        char    ProgBase[50];           /* file name for prog. base */

/*________________________________________________________________________
 * Startup Section
 */

        PRINT_HEADING;
        GET_FILE_NAME("KnowLedge Base : ",KnowBase);
        GET_FILE_NAME("Program Base   : ",ProgBase);
        Load_Information(KnowBase,ProgBase);

/*________________________________________________________________________
 *  Main Goal Achievement Section
 */

        for (;;)
        {
                User_Interface(Main_Goal,KnowBase,ProgBase);

                Goal_Information = Get_Goal_Information(Main_Goal);
                Assign_Inputs(Main_Goal,Goal_Information);
                Achieve_Goal(Main_Goal,Goal_Information);
/*              if (Goal_Information->Done == FALSE) Learn_From_User(Main_Goal,Go
        }

}
```

```
/***************************************************************************

Procedure Name   : Print_Files()
Part of          : Hierarchical Robot Control System
File Name        : pf.c
Date             : 05.15.86
Author           : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose          : To print all the data base files

***************************************************************************/

#include        "defs.h"        /* user defined goodies */

void Print_Files(KnowBase,ProgBase)
char    *KnowBase;              /* file name for knowledge base */
char    *ProgBase;              /* file name for program   base */
{
        FILE    *fd;            /* file descriptor */
        FILE    *fopen();       /* open function   */
        char    com[COMMAND_SIZE];

        printf("\n");

        if ((fd = fopen(ProgBase,"r")) == NULL) ERROR_MSG("Can't open ",ProgBase)
        printf("Program Base: %s\n",ProgBase);
        while (fscanf(fd,"%s!",com) != EOF) printf("%s\n",com);
        close(fd);

        if ((fd = fopen(KnowBase,"r")) == NULL) ERROR_MSG("Can't open ",KnowBase)
        printf("\nKnowledge Base file: %s\n",KnowBase);
        while (fscanf(fd,"%s!",com) != EOF) printf("%s\n",com);
        close(fd);
}
```

```
/***********************************************************************

Procedure Name  : Print_Goal_Result()
Part of         : Hierarchical Robot Control System
File Name       : pgr.c
Date            : 05.15.86
Author          : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose         : To print the goal and its input/output

***********************************************************************/

#include        "defs.h"        /* user defined goodies */

void Print_Goal_Result(Goal,Goal_Information)
char            Goal[];
goalinfo        *Goal_Information;
{

        int     i;
        input   *Input;
        output  *Output;
        char    Arg[COMMAND_SIZE];

        if (Goal_Information->OutOK == FALSE)
        {
                printf("Failed    : %s\n",Goal);
                return;
        }
        i = -1;
        Input = Goal_Information->Needed_Input;
        printf("Achieved  : ");

        while (Goal[++i] != '\0' && Goal[i] != EOC && Goal[i] != OUTP)
        {
                if (Goal[i] == ARG)
                {
                        printf("~%s~",Input->Value);
                        while (Goal[++i] != ARG) ;
                        Input = Input->Next;
                }
                else printf("%c",Goal[i]);
        }
        Output = Goal_Information->Needed_Output;
        strcpy(Arg,Goal_Information->Output);
        i = -1;
        while (Arg[++i] != '\0' && Arg[i] != EOC)
        {
                if (Arg[i] == ARG)
                {
                        printf("~%s~",Output->Value);
                        while (Arg[++i] != ARG) ;
                        Output = Output->Next;
                }
                else printf("%c",Arg[i]);
        }
        printf("\n");
}
```

```c
/*****************************************************************************

Procedure Name  : Print_Know_Table()
Part of         : Hierarchical Robot Control System
File Name       : pkt.c
Date            : 05.15.86
Author          : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose         : To print the knowledge base

*****************************************************************************/

#include        "defs.h"                /* user defined goodies */

void Print_Know_Table()
{
        extern Kelem    *Know_Table[27];/* know. base */
        Kelem           *Curr_Elem;
        subgoal         *Subg;
        int             i;

        printf("\nKnowledge Base:\n\n");

        for (i = 0; i < 27; i++)
        {
                Curr_Elem = Know_Table[i];
                while (Curr_Elem != NULL)
                {
                        printf("%s\n",Curr_Elem->Goal);
                        Subg = Curr_Elem->Goal_Info->SubGoals;
                        while (Subg != NULL)
                        {
                                printf("\t%02d %s\n",Subg->Group,Subg->SubGoal);
                                Subg = Subg->Next;
                        }
                        Print_Probabilities(Curr_Elem);
                        Curr_Elem = Curr_Elem->Next;
                }
        }
}
```

```
/***************************************************************************

Procedure Name  : Print_Probabilities()
Part of         : Hierarchical Robot Control System
File Name       : pp.c
Date            : 05.15.86
Author          : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose         :

***************************************************************************/

#include        "defs.h"                /* user defined goodies */

void Print_Probabilities(Curr_Elem)
Kelem   *Curr_Elem;
{
        printf("Goal Prob = %f\n\n",Curr_Elem->S/(float)Curr_Elem->N);
}
```

```
/*************************************************************************

Procedure Name   : Read_In_Probs(fd,Curr_Elem)
Part of          : Hierarchical Robot Control System
File Name        : rip.c
Date             : 05.15.86
Author           : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose          : To read in probabilities for a goal

*************************************************************************/

#include        "defs.h"                    /* user defined goodies */

void Read_In_Probs(fd,Curr_Elem)
FILE            *fd;
Kelem           *Curr_Elem;
{
        fscanf(fd,"%d%d",&(Curr_Elem->N),&(Curr_Elem->S));
}
```

```
/***************************************************************************

Procedure Name  : Remove_SubGoal()
Part of         : Hierarchical Robot Control System
File Name       : rs.c
Date            : 05.15.86
Author          : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose         : To remove subgoals in a tree

***************************************************************************/

#include        "defs.h"        /* user defined goodies */

void Remove_SubGoal(Goal_Info,Sgr)
goalinfo        *Goal_Info;
int             Sgr;
{
        subgoal *SubGoal;
        subgoal *Temp;
        subgoal *Next;
        boolean FLAG = FALSE;

        SubGoal = Goal_Info->SubGoals;
        Temp = SubGoal;

        while (SubGoal != NULL && Sgr != SubGoal->Group)
        {
                Temp = SubGoal;
                SubGoal = SubGoal->Next;
        }
        if (Temp == SubGoal) FLAG = TRUE;
        while (SubGoal != NULL && Sgr == SubGoal->Group)
        {
                Next = SubGoal->Next;
                cfree(SubGoal);
                SubGoal = Next;
        }
        if (FLAG)
        {
                if (SubGoal == NULL) Goal_Info->SubGoals = NULL;
                else Goal_Info->SubGoals = SubGoal;
        }
        else Temp->Next = SubGoal;
}
```

```
                        {
                                Inp->Next = MORE_MEMORY(1,input);
                                Inp = Inp->Next;
                        }
                        Inp->Inp_Name = MORE_MEMORY(1+no,char);
                        strcpy(Inp->Inp_Name,Argument);
                }
        }
}
```

```
/***************************************************************************

Procedure Name  : Store_Arguments()
Part of         : Hierarchical Robot Control System
File Name       : sa.c
Date            : 05.15.86
Author          : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose         : To store arguments for a goal that is read

***************************************************************************/

#include        "defs.h"        /* user defined goodies */

void Store_Arguments(com,Goal_Info)
char            com[];
goalinfo        *Goal_Info;
{
        output          *Out;
        input           *Inp;
        char            Argument[100];
        int             no;
        int             i = -1;
        boolean         OUTPUT = FALSE;

        while (com[++i] != '\0' && com[i] != EOC)
        if (com[i] == OUTP)
        {
                OUTPUT = TRUE;
                Goal_Info->Output = MORE_MEMORY(1+strlen(&com[i]),char);
                strcpy(Goal_Info->Output,&com[i]);
        }
        else if (com[i] == ARG)
        {
                no = 0;
                while (com[++i] != ARG)
                {
                        if (com[i] == '\0') ERROR_MSG("sa.c: Missing ~ when reading
                        Argument[no++] = com[i];
                }
                Argument[no] = '\0';
                if (OUTPUT)
                {
                        if (Goal_Info->Needed_Output == NULL)
                        {
                                Out = MORE_MEMORY(1,output);
                                Goal_Info->Needed_Output = Out;
                        }
                        else
                        {
                                Out->Next = MORE_MEMORY(1,output);
                                Out = Out->Next;
                        }
                        Out->Out_Name = MORE_MEMORY(1+no,char);
                        strcpy(Out->Out_Name,Argument);
                }
                else
                {
                        if (Goal_Info->Needed_Input == NULL)
                        {
                                Inp = MORE_MEMORY(1,input);
                                Goal_Info->Needed_Input = Inp;
                        }
                        else
```

```
/*********************************************************************

Procedure Name   : Store_Information()
Part of          : Hierarchical Robot Control System
File Name        : si.c
Date             : 05.15.86
Author           : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose          : To store information about a goal on file

*********************************************************************/

#include        "defs.h"                /* user defined goodies */

void Store_Information(KnowBase,ProgBase)
char    KnowBase[];
char    ProgBase[];
{
        int             i;                      /* loop counter */
        extern Kelem    *Know_Table[27];        /* know. base */
        Kelem           *Curr_Elem;
        subgoal         *Subg;
        FILE            *fd1;                   /* file descriptor */
        FILE            *fd2;                   /* file descriptor */
        FILE            *fopen();               /* open function   */

        printf("\nTrying to Store Knowledge Base . . . ");
        if ((fd1 = fopen(KnowBase,"w")) == NULL) ERROR_MSG("Can't open ",KnowBase)
        if ((fd2 = fopen(ProgBase,"w")) == NULL) ERROR_MSG("Can't open ",ProgBase)

        for (i = 0; i < 27; ++i)
        {
                Curr_Elem = Know_Table[i];
                while (Curr_Elem != NULL)
                {
                        if (Curr_Elem->Goal_Info->ProgNo == 0)
                        {
                                fprintf(fd1,"%d%s\n",Curr_Elem->Goal_Info->SubgNo,
                                Subg = Curr_Elem->Goal_Info->SubGoals;
                                while (Subg != NULL)
                                {
                                        fprintf(fd1,"%d%s\n",Subg->Group,Subg->Sub
                                        Subg = Subg->Next;
                                }
                                Store_Probs(fd1,Curr_Elem);
                        }
                        else
                        {
                                fprintf(fd2,"%d%s\n",Curr_Elem->Goal_Info->ProgNo,
                                Store_Probs(fd2,Curr_Elem);
                        }
                        Curr_Elem = Curr_Elem->Next;
                }
        }
        fclose(fd1);                            /* close file */
        fclose(fd2);                            /* close file */
        printf("Stored !\n");
}
```

```c
/*****************************************************************************

Procedure Name  : Store_Probs()
Part of         : Hierarchical Robot Control System
File Name       : sp.c
Date            : 05.15.86
Author          : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose         : To store probabilities for a goal on file

*****************************************************************************/

#include        "defs.h"                /* user defined goodies */

void Store_Probs(fd,Curr_Elem)
FILE    *fd;
Kelem   *Curr_Elem;
{
        fprintf(fd,"%d %d\n",Curr_Elem->N,Curr_Elem->S);

        fprintf(fd,"\n");
}
```

```
/*
 * sort_thetas.c -  sorts the thetas in descending order.
 *
 * Author:       J.K. Lee
 *               Computer Science Dept.
 *               University of Utah
 * Date:         Tu. May 6 1986
 * Copyright (c) 1986 J.K. Lee
 */

#include "defs.h"

/*****************************************************************
 * TAG( sort_thetas.c )
 *
 *     sort_thetas
 *
 */
void sort_thetas(theta1,theta2,theta3)
double  *theta1,*theta2,*theta3;
{
        int     i, j;
        double  theta[3], temp;

        /* sort thetas in descending order */
        /* vertices is recorded in the same order as thetas */
        theta[0] = *theta1;
        theta[1] = *theta2;
        theta[2] = *theta3;

        for (i = 0; i < 3; i++)
                for (j = i+1; j < 3; j++)
                        if (theta[i] < theta[j])
                        {
                                temp = theta[i];
                                theta[i] = theta[j];
                                theta[j] = temp;
                        }
        *theta1 = theta[0];
        *theta2 = theta[1];
        *theta3 = theta[2];
}
```

```
/*************************************************************************

Procedure Name   : User_Interface()
Part of          : Hierarchical Robot Control System
File Name        : ui.c
Date             : 05.15.86
Author           : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose          : To get a command (main goal) from the user via the
                   keyboard

*************************************************************************/

#include        "defs.h"        /* user defined goodies */

void User_Interface(command,KnowBase,ProgBase)
char    command[];
char    KnowBase[];
char    ProgBase[];
{
        int     i;

        for (;;)
        {
                PRINT_MENU;
                printf("\nCommand: ");
                for (i = 0; i < 70; i++) printf("_");
                for (i = 0; i < 70; i++) printf("\b");
                fflush(stdout);
                scanf("%s",command);
                switch (command[0])
                {
                        case '1':       Print_Files(KnowBase,ProgBase);
                                        break;

                        case '2':       Print_Know_Table();
                                        break;

                        case '4':       PRINT_SYNTAX;
                                        break;

                        case '9':       Store_Information(KnowBase,ProgBase);
                                        printf("\nBye bye . . .\n\n");
                                        exit(1);

                        default:        TOUPPER(command);
                                        return;
                }
        }
}
```

```
/***************************************************************************

Procedure Name   : User_Provide()
Part of          : Hierarchical Robot Control System
File Name        : up.c
Date             : 05.15.86
Author           : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose          : To get the input for a goal

***************************************************************************/

#include        "defs.h"        /* user defined goodies */

int User_Provide(SubGoal,Input)
subgoal *SubGoal;
input   *Input;
{
        char    value[50];
        char    Name[50];

        if (Input->Inp_Name1 == '\0') strcpy(Name,Input->Inp_Name);
        else strcpy(Name,Input->Inp_Name1);

        printf("Input missing for: %s\n",SubGoal->SubGoal);
        printf("Can't find input for: %s\n",Name);
        printf("Please specify .......\n");
        printf("%s = ? : ",Name);
        scanf("%s",value);
        Input->Value = MORE_MEMORY(1+strlen(value),char);
        strcpy(Input->Value,value);
}
```

```
/***************************************************************************

Procedure Name  : Update_Probabilities()
Part of         : Hierarchical Robot Control System
File Name       : upr.c
Date            : 05.15.86
Author          : Nils Thune

Copyright (c) 1986 - THUNE DATA (T.D.)

Purpose         : To update probabilities for a goal

***************************************************************************/

#include        "defs.h"                /* user defined goodies */

void Update_Probabilities(Goal,Goal_Info)
char            Goal[];                 /* command string */
goalinfo        *Goal_Info;             /* pointer to goal information */
{
        extern Kelem    *Know_Table[27];/* know. base */
        Kelem           *Curr_Elem;

        Curr_Elem = Know_Table[Goal[0]-'A'];

        while (!GoalComp(Goal,Curr_Elem->Goal)) Curr_Elem = Curr_Elem->Next;

        if (Goal_Info->OutOK) Curr_Elem->S += 1.0;
        Curr_Elem->N += 1.0;
}
```

```
/***************************************************************************

File Name       : defs.h
Date            : 05.16.86

Purpose         : definitions used in the robot control system program

***************************************************************************/



/*___________________________________________________________________________
 * Definitions of constants
 */

#define COMMAND_SIZE    100     /* length of commando size */
#define TRUE            1       /* boolean constant */
#define FALSE           0       /* boolean constant */
#define DONE            1       /* is schema DONE ? */
#define GO              2       /* schema is not DONE ? */
#define LEARN           3       /* schema should learn from user */
#define QUIT            4       /* quit trying to obtain a goal */
#define NOTOUTOK        -1      /* output is not OK */

/* Some constants used when loading/writing data bases  */

#define ARG             '~'     /* command ARGument     */
#define OUTP            ':'     /* OUTPut follows       */
#define EOC             '!'     /* End Of Command       */


/*___________________________________________________________________________
 * List all user/system defined include files here
 */

#include        <ctype.h>       /* macro goodies */
#include        <stdio.h>       /* input output goodies */
#include        <math.h>        /* math goodies */
#include        <strings.h>     /* string operation goodies */
#include        "struct.h"      /* structure definitions */
#include        "func.h"        /* function declarations */
#include        "macro.h"       /* include my personal defined macros */
```

```
/**********************************************************************

File Name       : func,h
Date            : 05.16.86

Purpose         : function declarations of programs used in the robot
                  control system

**********************************************************************/



/*____________________________________________________________________
 * Function declarations
 */

void            Achieve_Goal();
int             Achieve_SubGoals();
void            Assign_Inputs();
void            Assign_Names();
boolean         Check_Goal_Status();
void            Execute();
subgoal         *Find_Best_AND_group();
float           Find_Prob();
goalinfo        *Get_Goal_Information();
int             Get_Input();
void            Get_Output();
boolean         GoalComp();
void            Insert();
int             Learn_From_User();
void            Load_Information();
void            Load_Knowledge_Base();
void            Load_Program_Base();
void            Print_Files();
void            Print_Goal_Result();
void            Print_Know_Table();
void            Print_Probabilities();
void            Read_In_Probs();
void            Remove_SubGoal();
void            Store_Arguments();
void            Store_Information();
void            Store_Probs();
void            Update_Probabilities();
void            User_Interface();
int             User_Provide();
```

```
/**************************************************************************

File Name        : macro.h
Date             : 05.16.86

Purpose          : macros for the robot control system

**************************************************************************/


/*_____________________________________________________________________________
*
*       Macro that initializes an array to the 'value' given
*/

#define INITIALIZE(no,value,array) {int i;\
                                    for (i = 0; i < no; i++)\
                                        array[i] = value; }


/*_____________________________________________________________________________
*
*       Macro that prints out the heading for the robot control system
*/

#define PRINT_HEADING   {system("clear");\
                         printf("\t\tHierarchical Robot Control System\n");\
                         printf("\t\t\t   Release 1.30\n\n\n\n\n");\
                         printf("Please specify .....\n\n");}

/*_____________________________________________________________________________
*
*       Macro that prints out menu
*/

#define PRINT_MENU     {printf("\n\n\t\tM E N U\n\n");\
                        printf("1...Print Knowledge and Program files\n");\
                        printf("2...Print Knowledge Base\n");\
                        printf("4...Print Command Syntax\n");\
                        printf("9...Exit Control System\n"); }
/*_____________________________________________________________________________
*
*       Macro that prints the command syntax
*/

#define PRINT_SYNTAX    printf("\nSyntax: letters~argument~:letters~argument~!\n"


/*_____________________________________________________________________________
*
*       Macro that reads in a file name from the terminal
*/

#define GET_FILE_NAME(string,file) {\
                        printf("%s",string);\
                        scanf("%s",file); }


/*_____________________________________________________________________________
*
*       Macro which returns a pointer to a string of type 'type' of length
*       'size'
*/

#define MORE_MEMORY(size,type)  (type *)(calloc(size,sizeof(type)))
```

```c
/*______________________________________________________________________
 *
 *       Macro which converts characters in a string to upper case
 */

#define TOUPPER(array)   {int i = 0;\
                          while (array[i] != '\0') {\
                          if (array[i] == ARG) while (array[++i] != ARG) ;\
                          else if (islower(array[i])) array[i] = toupper(array[i]);
                          ++i; } }


/*______________________________________________________________________
 *
 *       Macro that prints out an error message and then exits the program
 */

#define ERROR_MSG(s1,s2){printf("\n\n%s %s\n\n",s1,s2);\
                          printf("Fatal error, bailing out . . . .\n\n");\
                          exit(-1); }
```

```
/**************************************************************************

File Name       : struct.h
Date            : 05.16.86

Purpose         : definitions used in the robot control system program

**************************************************************************/



/*___________________________________________________________________________
 * Type definitions
 */

typedef char    boolean;        /* boolean is the same as char, 1 byte */


/*___________________________________________________________________________
 * Structures
 */


typedef struct _Kelem
{
        struct  _Kelem          *Next;          /* Next goal in Kbase */
        int                     N;              /* # of trials */
        int                     S;              /* # of successes */
        char                    *Goal;
        struct  _goalinfo       *Goal_Info;

} Kelem;        /* element in hash table */

/*________________________________________________________________________*/

typedef struct _input
{
        char            *Inp_Namel;     /* name of input */
        char            *Inp_Name;      /* name of input */
        char            *Value;         /* acctual input */
        struct _input   *Next;          /* pointer to next needed input */

} input;        /* holds input information for a goal */

/*________________________________________________________________________*/

typedef struct _output
{
        char            *Out_Namel;     /* name of output */
        char            *Out_Name;      /* name of output */
        char            *Value;         /* acctual output */
        struct _output  *Next;          /* pointer to next needed output */


} output;       /* holds output information for a goal */

/*________________________________________________________________________*/

typedef struct _subgoal
{
        char                    *SubGoal;       /* holding the subgoal 'command'
        int                     Group;          /* group # */
        struct _goalinfo        *Goal_Info;     /* Pointer to subgoal information
        struct _subgoal         *Next;          /* Pointer to next subgoal */

} subgoal;      /* subgoal information for a goal */
```

```
/*_________________________________________________________________________*/

typedef struct _goalinfo
{
        boolean Null;                   /* if goal info was found or not */
        int     ProgNo;                 /* If program, this is the reference */
        boolean Done;                   /* Is it done ? */
        input   *Needed_Input;          /* Input needed to achieve goal */
        output  *Needed_Output;         /* Output needed to achieve goal */
        char    *Output;                /* Output string */
        boolean OutOK;                  /* if needed ouput is OK */
        subgoal *SubGoals;              /* Subgoals if goal is not program */
        int     SubgNo;                 /* number of subgoals */

} goalinfo;     /* holds information about goal achievement */

/*_________________________________________________________________________*/

typedef struct data
{
        int     hei;

} datanode;     /* */

/*_________________________________________________________________________*/
```

References

[1] J.S. Albus. *Brains, Behavior, & Robotics.* BYTE Publications Incorporated, 1981.

[2] J.A. Anderson. Cognitive and Psychological Computation with Neural Models. *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13(5):799-815, September/October, 1983.

[3] M.A. Arbib, T. Iberall and D. Lyons. *Coordinated Control Programs for Movements of the Hand.* COINS 83-25, University of Massachusetts, August, 1983.

[4] D.H. Ballard and C.M. Brown. *Computer Vision.* Prentic-Hall, Inc, 1982.

[5] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence.* Addiso-Wesley Publishing Company, Inc., 1981.

[6] M. Beeler. *Butterfly Parallel Processor Tutorial for Programming in the C Language.* BBN 6190, BBN Laboratories Incorporated, March, 1986.

[7] Bir Bhanu, Nils Thune, and Mari Thune. CAOS: A Hierarchical Robot Control System. *To appear in IEEE Transactions on Robotics and Automation* , 1987.

[8] Bir Bhanu, Nils Thune, Jih Kun Lee, and Mari Thune. Hierarchical Robot Control in a Multisensor Environment. *To appear in SPIE, Intelligent Robots and Computer Vision* , 1986.

[9] C.M. Brown, C.S. Ellis, J.A.Feldman, T.J. LeBlanc and G.L. Peterson. *Research with the Butterfly Multicomputer.* Technical Report, BBN Laboratories Incorporated, 1986.

[10] Joseph F. Engelberger. *Robotics in Practice.* ama com, 1980.

[11] J.A. Feldman. Connectionist Models and Parallelism in High Level Vision. *Computer Vision, Graphics, and Image Processing* 31(2):178-200, August, 1985.

[12] Arthur C. Guyton. *Human Physiology and Mechanisms of Disease.* W.B. Saunders Company, 1982.

[13] Frederick Hayes-Roth, Donald A. Waterman, Douglas B. Lenat. *Building Expert Systems.* Addison-Wesley Publishing Company, Inc., 1983.

[14] T. Iberall and D. Lyons. *Towards Perceptual Robotics.* COINS 84-17, University of Massachusetts, August, 1984.

[15] Peter Jackson. *Introduction to Expert Systems.* Addison-Wesley Publishing Company, Inc., 1986.

[16] C. Jacobus, W.D. Lee and J. Norton. Flexible Assembly and Inspection of a small Electric Fuel Pump. *SPIE, Intelligent Robots and Computer Vision* 579:528-536, 1985.

[17] Eric R. Kandel and James H. Schwartz. *Principles of neural science.* Elsevier Science Publishing Co, Inc., 1985.

[18] Robert R. Kessler. *Objective Lisp.* Brown and Company, 1986.

[19] Stephen W. Kuffler, John G. Nicholls, A. Robert Martin. *From Neuron To Brain.* Sinauer Associates Inc. Publishers, 1984.

[20] E.R. Lewis. The Elements of Single Neurons: A Review. *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13(5):702-710, September/October, 1983.

[21] D. Mankins. *Chrysalis Programmers Manual, Version 2.3.* BBN 6191, BBN Laboratories Incorporated, May, 1986.

[22] Amar Mitiche and J. K. Aggarwal. Multiple Sensor Integration/Fusion through Image Processing. *Optical Engineering* 25(3):380-386, March, 1986.

[23] D.S. Nau. Expert Computer Systems. *IEEE Computer* 16(2):63-85, February, 1983.

[24] K.J. Overton. *The Acquisition, Processing, and use of Tactile Sensor Data in Robot Control.* COINS 84-08, University of Massachusetts, May, 1984.

[25] B. Thomas, R. Gurwitz, J. Goodhue, D. Allen and M. Beeler. *Butterfly Parallel Processor Overview.* BBN 6148, BBN Laboratories Incorporated, March, 1986.

[26] B. Thomas. *The Uniform System Approach to Programming the Butterfly Parallel Processor, Version 1.* BBN 6149, BBN Laboratories Incorporated, March, 1986.

[27] Nils Thune. Spherical Control. *In preparation* , 1987.

[28] D.A. Waterman. *A Guide to Expert Systems.* Addison-Wesley Publishing Company, Inc., 1986.