L O G I C P R O G R A M M I N G F O R T H E L A W

DISSERTATION SUBMITTED FOR THE DEGREE OF

MASTER OF TECHNOLOGY

BY

W P SHARPE

BRUNEL UNIVERSITY

JUNE 1984

ACKNOWLEDGEMENTS

I am grateful to Bob Kowalski, Peter Hammond and Marek Sergot for introducing me to the field of logic programming and the law, and for their help in providing source material and discussing problems. I would like to thank Tom Addis for his very helpful comments on the first draft of this report. I am especially grateful to Susan Fuller who typed it all to meet impossible deadlines. I am also happy to acknowledge the support of the Science and Engineering Research Council.

SUMMARY

This project looks at the use of logic programming for building intelligent knowledge based systems. The domain of law is chosen as a vehicle for the study which has three main sections. First, the field of knowledge representation is surveyed in order to put logic in the context of other formalisms. Secondly, the current state of the art in intelligent legal systems is reviewed, with particular attention to recent systems implemented in PROLOG. Lastly, a practical system to represent a piece of legislation is described in detail. This study shows the strength of logic as an analytic tool for structuring knowledge and as a tool for building knowledge based systems. It also shows, however, that the analysis of knowledge for mechanical representation is still in its infancy. Simplistic translation of explicit knowledge into a logic program produces a system of no significant power; a number of structuring principles must be used to organise the domain knowledge into a problem solving system. The discovery of these principles is the subject of the infant science of knowledge engineering. This project suggests a few such principles for the domain of written law.

C O N T E N T S

CHAPTER I

LOGIC PROGRAMMING FOR THE LAW

INTRODUCTION

This project is a study of the applicability of the techniques of logic programming to problems in representing legislation for the purposes of intelligent consultation. The motivations for the choice of domain and technique are both practical and theoretical.

The body of statute law in force in this country has been estimated to run in excess of 25 million words. The law affects us all in many different aspects of our lives and the problem of obtaining accurate knowledge of the law is becoming acute. Even solicitors are concerned that they are not able to keep up to date with the law that is relevant to their clients. When case law is added in, the body of material relevant to our daily affairs becomes quite unmanageable. The practical case for some automatic assistance with managing this body of knowledge is therefore clear.

Artificial Intelligence as a discipline has for a long time studied the problems of assisting humans in problem solving in domains where the concepts are of great complexity, such as medicine or mineral prospecting. Of recent years the recognition that knowledge of a domain and knowledge of problem solving methods to use in that domain are intimately related, has led to the invention of the terms 'knowledge based programming' and 'knowledge engineering' to represent this subset of Artificial Intelligence. These terms are used to describe the processes of building what are generally known as expert systems in which

expert knowledge of a domain is represented in some explicit way. What makes this style of programming different from conventional programming is the use of representational formalisms that bear a natural relationship to the knowledge to be expressed. The important aspect of these formalisms is that they have explicit means for representing inferential associations and other knowledge structuring concepts absent from the traditional languages. The process of developing an expert system is then called knowledge engineering because the expert is 'simply' making explicit his knowledge of his domain in the particular formalism.

This is contrasted with traditional programming in which the problem must be translated into an expressive form of data structures and procedures whose semantics are more to do with the state changes of a computer than the problem to be solved.

The development of languages whose semantics are more closely related to problem solving than to machine operations is an area of active research that goes wider than artificial intelligence and is generally now known as the field of declarative languages. This term indicates that the languages describe what should be done not how it should be done by a computer. These languages are strictly declarative with respect to the underlying implementation, but when used for knowledge representation they may describe either assertions or problem solving procedures. The important point is that the procedures are expressed in terms of domain heuristics, not manipulation of internal data structures of the kind we are familiar with in traditional computing. This point is dealt with in detail in this study.

Amongst the declarative languages there are two main classes: functional languages and logic languages. Within AI, LISP is the functional

language most widely used. In practical use its many non functional features are also extensively exploited. It is very powerful for building and manipulating complex data structures in an efficient manner. Functional languages however do not have "built-in" inferential mechanisms of the kind required for knowledge engineering. These are found in the logic languages, which conversely lack the myriad procedures for handling data structures found in LISP.

A recent definition of intelligent knowledge based systems is that they are ones which "apply inference to knowledge to perform a task" [1]. While no general definition of machine intelligence can hope to gain total approval in the current state of our understanding, there is widespread acceptance of inference and knowledge representation as cornerstones of the subject. That we seek to understand basic concepts by trying to model natural phenomena through the manipulation of symbols is not peculiar to the science of artificial intelligence. In the case of AI the phenomena are tasks which, if performed by natural agents, we think of, in some loose sense, as requiring 'intelligence'. This is taken up in more detail in the next chapter. Given that inference is a fundamental part of AI we will naturally be interested in programming languages which have powerful inferential mechanisms. In the logic programming languages these mechanisms are combined with the development of the declarative approach to programming; the use of these languages for knowledge engineering thus becomes of interest both from the perspective of AI and the development of declarative programming in general. The claim is made for logic programming that its declarative semantics are a natural and powerful tool for knowledge representation, and that its procedural semantics confer upon it the ability to apply the knowledge represented to perform tasks. The purpose of this project was to study that claim.

Beyond the practical reasons for choosing the legal domain for this study the theoretical justification lies in the apparent correspondence between the written expression of the law and the formalism of logic programming. The law is one domain where, superficially, there is no problem with the acquisition of knowledge. What is written down is the law, and it is written down in the form of rules "if ... then ...". The Horn clause subset of logic is a formalism that expresses rules directly. We therefore have a match between a domain and the formalism we wish to investigate. Given this initial match we may feel justified in expecting that any limitations found here will also be found in domains where the match does not exist. We find in this study that the match is superficial only and perhaps tends to conceal the relationship between domain knowledge and a representational language. In the second chapter of this report we survey the subject of knowledge representation in general and find that this relationship is subtle and still largely not understood. This survey puts the use of logic in a wider context and goes some way to establishing its role in the study of knowledge based programming.

The law is put to many different uses by many quite different classes of users: the ordinary citizen would like to be able to have definitive advice on his entitlement to some benefit, say, or know in rough detail what his rights are in some situation; an employer wishes to know precisely what are the various procedures he must comply with in respect of his employees; a solicitor needs very detailed knowledge of all the laws pertaining to his client's case and, more importantly, knowledge of how to apply the law, the procedure for its use; the legal drafter needs assistance with constructing complex written documents that convey the intended meaning; the policy maker needs to be able to model the effect

of a hypothetical piece of legislation. At the heart of all these uses there is one and the same law. We would like to understand how that legal knowledge should be structured so that it can be put to all these uses, and where the boundaries are between a representation of the law in general and its representation for a particular task. Chapter 3 looks at the existing literature on the application of AI to the law and reviews the state of our current knowledge of these questions.

A major part of the time spent on this project was devoted to a practical investigation into the use of logic programming for the law. This investigation took the form of building a system capable of answering a restricted set of queries on a very small amount of legislation. This study provided the insights upon which the rest of the survey was built. This part of the project is reported on in detail in Chapter 4.

In the final chapter the conclusions of this investigation are presented with some suggestions for future lines of work.

* * * *

CHAPTER II

KNOWLEDGE REPRESENTATION

2.1 INTRODUCTION

The subject of knowledge representation is an area of current research that displays an immense variety of approaches and intellectual positions. It would be possible to survey it through the classic taxonomy of representational methods but that approach is not followed here. The review of current work contained in [7] shows that although some taxonomic classification is possible at a surface level into such methodologies as logic, production rules, semantic nets etc, there is deep disagreement about the very language in which the research problems should be stated. The authors of the survey were themselves surprised at the almost total heterogeneity in the replies to their questionnaire and are unable to draw out any general principles. Given such a situation, where each researcher takes a different attitude to the significance of even an elementary taxonomic classification of techniques, this study chooses to take a different approach. Newell [26] has made an attempt to give a new framework to the field of knowledge representation. This work will be taken as a reference point for the whole field against which the contributions of a number of other workers will be described. The papers chosen for review represent a wide spectrum of attitudes but, given the diversity already referred to, any claim to a comprehensive treatment of the subject (or the literature) must be unfounded.

In order to provide a framework for this discussion the argument of the paper by Newell is given in detail in the next section.

2.2 "THE KNOWLEDGE LEVEL" [26]

The particular contribution made by Newell is to propose a description of intelligent systems in terms of two levels - a knowledge level and a symbol level - in place of a single symbol level which is the prevalent approach. This approach is intended to remove dispute and clarify the methodology of AI, the paper itself does not aim to offer any radically new solutions to the problems of designing intelligent systems.

In order to place the notion of knowledge level in context Newell reviews the concept of a level of systems abstractions as it is found in such treatments as [41]. A level in a computer system is a description of a virtual machine. There is a medium on which a number of components operate. The components are built up according to a number of basic structural laws of composition to produce a system (virtual machine) whose actions may be described by certain laws of behavior. In a real system a level of abstraction must be defined in terms of the level below. It is important to realise that in this sense a level is an artefact and not a more abstract (in the sense of containing less irrelevant detail) description of what lies beneath. A given level can be used to realise a whole class of systems at the next higher level, but will in general also place a number of constraints on those systems. Each level is thus a specialisation of the one beneath and its existence is open to empirical observation and subject to the technological constraints of the level beneath. The second important characteristic of a systems level is that, once defined, it may be used as a tool for analysis and design essentially without regard to the lower levels. It

is in this way that it may be called a level of abstraction since the language of one level is used independently of its realisation, and describes behaviour in terms that need have no one-to-one functional counterpart in the components of the lower level. This of course is an idealised view, and Newell observes that in real computer systems a level is only one approximation, the constraints of one level propagate into the design at the next higher level and must be taken account of by the system designer.

Parenthetically we note that the driving moitivation behind the work in the declarative languages is the desire to construct a systems level which would allow design to proceed in a language having formal world semantics rather than one that is concerned with the manipulation of states within a computer. Newell's distinction of symbol and knowledge level helps us to understand where the bridge between the representation of external knowledge of the world and knowledge of representation and processing must be built. This point is returned to later.

The next step in the development of Newell's hypothesis is the assertion that the description of intelligent systems may be usefully made in terms of a functional decomposition.

Without attempting to offer a definitive decomposition he suggests a few basic functional components. The definition of a task enters through a perceptual component and is stored in an internal representation. Drawing on a goal structure and some general knowledge activity proceeds to manipulate the representation until a solution is available. The

representation, viewed in this way, is plainly just the component that confers on the system some degree of competence. We understand that the structures of the representation will be manipulated by the processes in a way that is consistent with a representational view of those structures. Newell observes that while the notion of a representation is used fairly precisely within computer science, competence or knowledge appears to be whatever it is that a representation has. This observation of actual practice is elevated by Newell to become his Knowledge Level Hypothesis:

"There exists a distinct computer system level, lying immediately above the symbol level, which is characterised by knowledge as the medium and the principle of rationality as the law of behavior".

Although Newell proposes a knowledge level which can therefore be described in terms applicable to any other systems level, he observes characteristics that distinguish it from other levels in all major respects. Firstly he observes that the structure of the knowledge level is very simple, and that variety at this level is a result of what is known rather than of structural complexity. Thus the components are:

- a physical body with arbitrary modes of interaction with the environment

- a body of knowledge, defined without regard to any constraints on internal structure or representation.

- a set of goals, only distinguished from the rest of the knowledge in respect of their function.

There are no structural laws for the composition of these components, they cannot be built up into more complex agents. We may note that Newell is here offering a different view of a systems level. In the earlier section we noted that a level is an artefact, and as such not a general abstract level of description of the external world. Here, on the contrary, we appear to be offered an epistimological framework for knowledge and rational behaviour of agents within real world environments. This conflict is perhaps resolved by the development of the concept of the knowledge level within a system as not having any extensional reality but being only intensionally expressed by the representation within the symbol level. This is nothing new in the study of science in general. It is only in AI, where the subject is the representation of the reasoning process itself, that there has been a particular problem in separating out the properties of a symbol system from its real world semantics. Newell notes the tendency in AI to make a mythology of knowledge representation, making it the locus of intelligence.

The law of behaviour of the knowledge level is the principle of rationality:

"If an agent has knowledge that one of its actions will lead to one of its goals then the agent will select that action".

This level of description asserts a global principle that governs the behaviour of the symbol level without asserting any mechanistic principles for its realisation. It is, following the previous paragraph, an empirical observation of the principle which we take to be appropriate

to the description of intelligent systems. Newell here is clearly giving concrete expression to the way in which AI design has proceeded, and he also meets the work of other disciplines such as utility theory, experimental psychology, decision theory, etc where the derivation of behaviour from goals is a central theme. Newell explicitly brings out this connection and also extends the principle with auxiliary ones which define rational behaviour under conditions of multiple simultaneous goals, etc. These principles are offered only as examples to explain the idea of knowledge level rather than as a set of carefully thought out and substantiated definitions. It is clear that these principles do not give us much insight into the definition of rational behaviour, but they do serve to distinguish the definitional task from the representational one.

He is at pains to point out that the investigation of these principles can never be complete, that the knowledge level unlike other system levels has a radical incompleteness. This means that sometimes the behaviour of a system cannot be entirely specified by the knowledge level but only in terms of the symbol level which realises it.

These considerations lead up to Newell's functional definition of knowledge as:

"Whatever can be ascribed to an agent such that its behaviour can be computed according to the principle of rationality".

This functional view allows us to ascribe to a system the essentially unbounded set of propositions that may be made as a result of knowledge about (competence with respect to) some aspect of the world. We have an

intuitive notion that a finite representation can intensionally hold knowledge that in its extensive form is unbounded, and indeed this intuition can be seen to be fundamental to a recent defintion of intelligent knowledge based systems as ones "which apply inference to knowledge to perform a task" [1]. An intelligent system generates by inference knowledge (propositions) that are relevant to the task in hand.

Given this functional defintion the problem for an intelligent agent is to create a symbol level that can solve the functional equation. The knowledge level. provides only a definition, not a theory, of representation. At the knowledge level we have rationality and knowledge producing behaviour; at the symbol level we cannot expect these to resolve in any predetermined way into data structures and processes. Representational theory is thus a separate domain of research from the definition of the knowledge level. Newell summarises the reduction of the knowledge to the symbol level in the following table:

Knowledge Level	Symbol level
Agent	Total symbol system
Actions	Symbol system with transducers
Knowledge	Symbol structure plus its proceses
Goals	(Knowledge of goals)
Principle of rationality	Total problem solving process

Having made this distinction between the levels it becomes possible to examine the contribution made by an AI system or methodology to each level separately. In doing so we shall find, as Newell observes, that the relationship between the levels is not pure because of the radical incompleteness of the knowledge level referred to earlier; and also because empirical observation of intelligent agents (psychology) shows how processing limitations intimately affect the realisation of the idealised knowledge level definition. In looking at particular AI systems Newell finds that in general they make their main contribution to one or other level rather than to both. Thus MYCIN [34], which is discussed in more detail in a later section, is a contribution essentially to the knowledge level, its processing regime being quite straightforward.

Newell relates his distinction of levels to that between epistemological adequacy and heuristic accuracy. Central to Newell's thesis is the propostion that a formalism and analysis that achieves epistemological adequacy with respect to some knowledge is not therefore bound to be heuristically accurate at the symbol level. In other words it is the distinction between using a tool to define the representation and using it to encode it. We should not expect a priori that any formalism will be appropriate to both uses. The view taken by certain workers, such as McCarthy, that logic is appropriate to both uses Newell asserts to be mistaken. He cites the now well established limitations of uniform proof procedures in defence of this and also observes in McCarthy's move towards distinguishing the concept of a number from its representation an implicit recognition of his own position.

Newell ends by speculating that his theory may allow us to build a generative class of rational goal directed systems; the design form being derived from the knowledge level as a universal description of intelligent systems.

In summary therefore the recognition of the knowledge level allows us to make the following assertions.

- knowledge is abstract and is present only intensionally in the structures and processes of the symbol level,

- tools for analysis of the knowledge level are distinct from the technologies of the symbol level;

- knowledge is a radical approximation, and an adequate model of an agent will include some description of its symbol level.

2.3 "ON THE EPISTEMOLOGICAL STATUS OF SEMANTIC NETWORKS" [6]

Brachman, like Newell, through a survey of historical practice in AI is lead to suggest an understanding of knowledge representation in terms of levels and to define a new level to clarify earlier confusions. In Brachman's case the subject is not knowledge representation in general but specifically the use of semantic networks as a representational tool.

A network is simply a collection of nodes joined by arcs and it becomes a representation of meaning through the association of some class of concepts with the nodes and relations with the links. That there has been no consistency between workers on the choice of concepts to attach to nodes and links is immediately apparent. Brachman suggests that the uniformity of the tool has also lead to an unfortunate tendency to confound representations that are essentially distinct. His reconstruction of the field identifies four levels that have been widely used and defines a fifth (the epistemological) that he beleives has not been adequately recognised or studied. These levels are:

- implementational
- logical
- epistemological
- conceptual
- linguistic

When isolated in a pure form a level is comprised of a number of primitives and an interpreter that processes them. A network is then a means of structuring primitives and the interpreter is a means of deploying the information contained in that structure. It should be noted that Brachman is here making it clear that the knowledge represented in a network is only adequately described by both the network and its interpretive processes.

In order to achieve the reconstruction into levels Brachman characterises a network level as having neutrality, adequacy and semantics. A level is neutral towards the level above it insofar as it does not force any

particular choice of primitives into it. It is adequate if it is sufficiently rich to represent all the semantics appropriate to its level. A level therefore requires that such a semantics should exist and be well defined.

The implementational level is found in the work of Nash-Webber and Reiter which treats a network as little more than a data structuring device of nodes and pointers for the construction of higher level logical language. As such it has nothing to say specifically about knowledge structuring or representation and has semantics no different from (say) a list processing language.

A network constructed at the logical level represents logical relationships. Nodes represent predicates and propositions and links represent logical relationships such as AND, SUBSET, etc. For this level the notion of adequacy is well defined being derived from the predicate calculus. From this point of view the semantic network may be seen as simply an alternative syntax for predicate calculus with the useful addition of organisational principles over normally unindexed predicate calculus statements.

The epistemological level is the focus of Brachman's study and is concerned with the relationship between the parts of an intension to the intension as a whole, and one intension to another. It describes the formal structure of conceptual units and their relationship independently of the knowledge they contain. This level is therefore about the definition of knowledge structuring primitives (such as property inheritence) rather than particular knowledge primitives that are found at the next higher level.

The conceptual level is typified by the work of Schank [30, 31]. At this level the designer describes case structures with their attendent cases or "slots". A node is associated with a case structure which defines some primitive piece of knowledge. For instance a verb structure has cases to define agent, object, etc. In Schank's work there are "primitive acts" with cases such as "instrument" and "direction". The primitives at this level are therefore directly concerned with discovering underlying unity in word senses and their case relations, ie with a framework for the meaning of language. It does not explicitly account for the internal structure of these senses.

The top most level, the linguistic has primitive elements which are language specific. The only example found by Brachman is OWL [16] although the view that language may be inseparable from the structuring of knowledge has been considered elsewhere [20]. In this view it is the knowledge itself which forms the structure, and the meaning of links cannot be asserted separately from the knowledge embodied in the network.

In discussing Brachman's analysis from the Newell perspective we need to distinguish the analysis itself from any particular system that he takes as an example. The analysis into five levels may then be seen as a contribution to our understanding of the knowledge level since it allows us to be more precise about the analytic tools used to describe knowledge. As pointed out above Brachman's analysis assumes that network formalisms are to be considered together with their interpreters and this gives them the competence like character of the knowledge level. Each level then attacks the notion of competence in a different way and it must be a subject of research whether the insights gained prove to have

value. Clearly Brachman's attitude is different to Newell's insofar as he does not ascribe to the logical level any primacy as an analytic tool, sees it in fact as subserviant to higher level analyses.

If we descend from the analysis to any particular network formalism then it is an open question whether we find the contribution being made at the knowledge or symbol level. Newell himself discusses one example at the conceptual level, that of Schank's conceptual dependency structures and places it within the knowledge level. Schank postulates a simple model for a language-free representation of meaning for "overt" physical or mental activities. Each conceptual dependency (CD) structure consists of a primitive act and a set of conceptual cases. Inference rules are grouped under the primitive acts. The number of primitive acts is quite small (eleven in [31]). Examples of acts are:

ATRANS	Abstract transfer of possession, ownership or control of a physical object
INGEST	Bring a substance into the body of a person or animal

The acts take place in a world of states, objects and actions. Actors have mental states that can participate in this world and have causal effects upon it. The dynamics of the model are summed up by Newell as follows:

World: states, actions, objects, attributes

Actors: objects with mentality

Cause:

An act results in a state

A state enables an act

A state or act initiates a mental state

A mental act is the reason for a physical act

A state disables an act.

The essential observation is that this model has been effectively implemented in a program (MARGIE) capable of an interesting ability to rephrase and perform inferences on natural language but that the implementation (the symbol level) did not shed any new insights on the model.

The semantic network formalism of Phillips [28], discussed further below (section 2.7) was designed, like Schanks, to help extract meaning from natural language text. Phillips not only discusses the relationship between his paradigmatic structures and the conceptual parser of Schank's model, thus adding to our understanding of that model, but also analyses the capabilities of the net interpreter in considerable detail. He is able to show that certain classes of meaning recognition can be achieved by path tracing processes, ie using a Chomsky type 3 grammer of the networks. Recognition of a class of more structurally complex concepts however requires pattern matching (equivalent to accepting a string of the form $a^n b^n$) and cannot therefore be accommodated by a path tracing

interpreter. This is a contribution specifically to our understanding of the symbol level since it relates to the complexity of implementation of a conceptual scheme that is independent of that scheme's justification at the knowledge level.

2.4 "THE EPISTEMOLOGY OF A RULE-BASED EXPERT SYSTEM" [9]

In this paper Clancey derives a general epistemological framework for the design of expert systems through a consideration of the knowledge that must be made explicit in order for an expert system to be useful for communicating its expertise. The system which was the subject of his analysis was MYCIN [34], the well known diagnostic and treatment advisor for bacterial infection. It was a reasonable hypothesis that a system able to apply an expert's knowledge of a domain, with some level of explanation already available, should provide a basis for communicating that expertise to the non-expert. The conclusion of Clancey's study is that MYCIN cannot communicate its expertise because several different kinds of knowledge are combined inextricably in the internal representation, so although the system may be said to have diagnostic knowledge (it performs that task) it has not got any knowledge of the diagnostic task itself. Clancey traces this shortcoming to the uniformity of the representation in production rules but observes that once the epistemology is clarified the actual representational notation is irrelevant. He is able to carry his analysis over to a number of different expert systems which taken together use all the common notations and thus we should understand this as a contribution purely to

the knowledge level. It is to be expected that there is much to be learnt about the perspicuity and power of the different notations to achieve the structuring he proposes. Although the original motivation of his study was to use a knowledge base for tutorial purposes his framework offers the possibility of increasing the power and flexibility of systems in performing their primary task.

The three types of knowledge that Clancey identifies within MYCIN and which he proposes should be made separately explicit are strategic, structural and support. Support knowledge justifies the use of a rule by giving it a basis in facts about the domain or the world in general. Strategic knowledge is concerned with plans for problem solving and lies above individual goals and hypothesis; to some extent it can be stated in domain independent terms. Structural knowledge defines abstractions that index the domain knowledge and thereby provide "handles" for the use of strategic knowledge. These three types of knowledge are now considered in more detail.

Rule justification is the process of logical argumentation to support the implication from a rule's antecedent to its consequent. In MYCIN Clancey distinguishes four kinds of justification: identification, world fact, domain fact, and causal. The first three of these are characterised by a degree of self-evidence within the problem domain that makes further support for the explanatory process unnecessary. For example, a domain fact rule is: "if a drug is administered orally and is poorly absorbed in the GI tract, then the drug was not administered adequately". Unravelling the justification of this rule would not add anything to the

understanding of the diagnostic domain and would have to draw on knowledge outside that domain. Such justification would properly belong in a system such as INTERNIST which is a more general advisor on internal medicine.

A simple example of a causal rule is "if a patient is less than 8 years old, don't prescribe tetracyline". The rule does not mention the underlying causal mechanisms upon which it rests (chelation - drug deposition in developing bones, causing blackened permanent teeth). In order to provide a satsifactory explanation one can imagine a tree of rules refining each step of the process to an ever greater level of detail. Clancey points out however that an explanation satisfies when it makes contact with known concepts, and from this point of view the explanatory process is one of generalisation from the specific unfamiliar detail to the general known class. The model for explanation may then be shown diagramatically as follows:

Explanation:	Immunosuppression	⇒	gram neg rods & entero bacteriaceae
Explanatory level:	Abstractions of problem features	⇒	abstractions of causal process
Rule level:	Antecedent	⇒	Consequent
Rule:	If white blood count less than 2.5	⇒	e.coli (.75) causing infection...

The abstract explanatory level is directly related to causal models for the process that is the subject of the diagnostic analysis. Making this causal model explicit then as a basis for explanations clearly has potential for giving the expert system recourse to general process knowledge when specific rules fail. Now Clancey observes that we would be quite mistaken to conclude from the discovery that a process model is necessary for explanation that MYCIN rules are written at the wrong level. The rules are good heuristics because they combine the knowledge of the explanatory level with strategic knowledge to drive the diagnostic procedure. The causal model is not an efficient subgoal structure for solving the diagnostic problem; it can justify the relations found between problem features, but those features are often deduced rather than presented. The model thus provides feedback that the diagnosis 'fits' and is a source of 'first principles' when heuristics fail.

Turning them to strategic knowledge we find that it is concerned with how to order goals and subgoals, choose between alternative paths of investigation etc. It is well known that good human problem solvers have efficient means for structuring their approach to the large amount of information and possible deductive paths present in a complex domain; Clancey concludes that some of these means can be expressed in domain independent terms. As examples:

- common (frequent) causes of disorder should be considered first

- if there are unusual causes then pursue them.

Clancey finds such strategic knowledge encoded into metarules in MYCIN, but there the domain independent strategy is implicitly present in a domain specific rule. To achieve the separation, and hence explicit representation it is necessary to include also the structural knowledge of the domain. In the above example, structural knowledge in MYCIN would include classes of 'common' and 'unusual' causes for particular features. The strategic knowledge is procedural in nature and Clancey places the issue of integrating the domain specific heuristics with the procedural, strategic knowledge (for problem solving as opposed to explanation) at the heart of the 'declarative/procedural controversy' [42].

The structural knowledge, if it is to be consistent with the explanatory knowledge of the system, must pertain to the same relations for hierarchically abstracting data and hypothesis as were discussed under the heading of support knowledge. From an examination of a number of expert programs (DENDRAL, AM, etc) Clancey is able to abstract a number of structuring principles that provide handles for the strategic process. In the following examples KS stands for 'knowledge source' and means an inference association:

- organise KSes for each hypothesis on the basis of how KS data relates to the hypothesis, for focusing on problem features (c.f NEOMYCIN)
- organise KSes hierarchically by hypothesis for consistency in data-directed interpretation.

Strategic rules might then be:

- Do not consider KSes that are subtypes of ruled-out hypothesis.

- Consider KSes that abstract known data.

What Clancey wishes to stress is that by keeping the strategic knowledge independent the structure of the supporting knowledge can be made explicit and hence accessible to an explanatory system.

Clancey believes that this analysis is a useful basis for the design of new expert systems and is independent of the notation used for its representation. Production rules were used in his reconstruction of MYCIN into NEOMYCIN to demonstrate these principles. He sees the design process as essentially cyclic, in which changes are made to the prototype rules until a new epistemological pattern emerges leading to a redesign of the rule set.

2.5 "FRAME REPRESENTATIONS AND THE DECLARATIVE/PROCEDURAL CONTROVERSY" [42]

Winograd first of all sets out the declarative and procedural positions on knowledge representation. The declarative approach asserts that knowledge can be stated and represented without reference to the uses to which it may be put. Competence then rests separately upon a set of facts for a domain and a set of procedures for manipulating facts of all sorts. Such a view, if used as a representational methodology, gives to a knowledge base an understandability and flexibility: declarative statements are a prevalent form of communicating knowledge, and the knowledge base can be modified incrementally by the addition of assertions.

For the procedural approach Winograd cites three particular advantages. Firstly, he observes that many of the things we know are in fact best

seen as procedurs, eg manipulations in a blocks world. Secondly, procedures can express more naturally second order knowledge about the use of declarative knowledge, eg "the relation NEAR is transitive as long as you don't try to use it too many times in the same deduction". Thirdly, he ascribes to procedures the ability to hold strategic knowledge, using this term to mean the domain dependent integration of Clancey's strategic and structural knowledge, ie "if you are trying to deduce this particular sort of thing under this particular set of conditions, then you should try the following strategies." Since this knowledge is concerned with the control of the deductive process and entails the use of domain specific knowledge Winograd concludes that a procedural description is more natural.

Winograd finds the source of the dispute about the relative merits of the two approaches in different views on the question of modularity in system construction. The declarative view allows a strong independence between "what" and "how" and confers learnability and understandability. The procedural approach allows more powerful interaction between the "chunks" of knowledge and allows them to enter into the control of this reasoning process. In particular expressions of both views he observes a move towards the other. Thus production systems are moving AI systems away from the general power of procedural interaction towards modular interaction through a structural database. In the other direction Sussman [40] imports domain knowledge into the general problem solving procedure to guide the backtracking process.

The attempted synthesis of these approaches suggested by Winograd is based almost entirely on a representational method rather than an

epistemological analysis. The representational format is called a "frame". A knowledge representation is then to be built up from frames arranged in a generalisation hierarchy. This is a structure of isa links connecting concepts to those of which they are specialisations. This hierarchy is operationally a hierarchy of descriptions in which additional properties are added as one descends the isa links.

A frame holds the internal information about a node in the hierarchy; this information is recorded in "slots" or components which identify the important elements (IMPS). These IMPS are themselves other frames and the links between frames established in this way are distinct from those of the isa hierarchy. Winograd is drawn into the implementational issues when he decides that these pointers should be able to be a path of IMP names to address an element held in another frame.

Having introduced a uniform notation Winograd explicity rejects the notion that it should be given a general, uniform interpreter. The purpose of the notation is to facilitate procedural attachment, ie to incorporate algorithmic knowledge into a modular, declarative organisation. From a study of the examples in the paper it is clear that all that Winograd is offering is a technological device to mix general deductive schemes (eg isa hierarchies) with domain specific ones. The notation does not offer us any useful insights into the epistemological questions of how the general deductive framework should be designed (c.f. Schank, Phillips ibid). The paper was written before the use of deduction systems to perform computations became established and with the benefit of hindsight we can see that the issue was really about how to use domain information to control the deductive process. The invention of languages with both a procedural and declarative semantics essentially

removed the declarative/procedural controversy in the form expressed here. The issues of allowing domain information to control the deductive process are taken up in the following section. Here we can note that the frame notation has continued to be developed, no longer as a solution to a procedural/declarative controversy but in the ways briefly discussed in section 2.7 as a concept-grouping and hypothesis generation device.

2.6 "LOGIC FOR PROBLEM SOLVING" [21, 22]

Logic programming is the technique of combining the expression of a problem in logic with an automatic proof procedure in order to produce a problem solver. The technique came of age with the machine oriented formalism of first order logic called Resolution [29], and now has its best known expression in the Prolog system. Prolog is restricted to problem definitions expressed in the Horn clause subset of logic; we return to this later in this section but the remarks that follow apply to logic programming in general.

At its simplest, a logic program is a collection of assertions and rules. A rule is of the form:

A if B and C .....

Typically the expression of a problem solving task in this form falls into three parts [21]:

(1) Assertions and rules which describe the problem domain in general.

(2) Problem specific assertions expressing the hypothesis of the problem to be solved.

(3) A goal statement which expresses the problem itself.

The problem is solved by the application of general inference techniques to (1) and (2) to derive (3). Now this addition of a proof technique to the declarative problem statement gives to the rules in (1) a procedural interpretation. If we have a rule of the form:

A if B and C

and the proof procedure is that of Prolog (left to right, depth first with backtracking) then the procedural interpretation of the rule is:

(1) reduce problem A to subproblems B and C
(2) solve B
(3) solve C
(4) if C fails, backtrack and attempt to resatisfy B. Solve C.
etc

The procedural semantics thus allows a logic programmer to describe a problem solving strategy as well as a problem specification. It is important to realise that this strategy does not reside in either one component (logic or control) of the logic programme. The problem specification must be designed with a view to its pragmatic application

by the particular control strategy that is available. This perspective on problem solving lead Kowalski to propose the slogan equation:

Algorithm = Logic + Control

This expresses the idea that the statement of what an alogorithm does may be kept separate (in a logic program) from a control component which affects only efficiency and not the meaning of the algorithm. Kowalski argues that an algorithm expressed in this way provides two distinct means for improving its efficiency. Either the problem representation can be changed to specify the problem in a new way, or the problem solving (theorem proving) capabilities of the program executor can be improved. On the latter possibility Kowalski notes [21] that a completely satisfactory autonomous control strategy has not yet been designed and that a number of languages have been developed to give the programmer more flexibile explicit control (eg PLANNER [19]). The development of a problem specification is further developed in [22].

In [22] Kowalski takes as an example the fifteen puzzle of sorting the initial state.

2	10	6	5
13		3	12
1	14	9	8
4	15	11	7

to the goal state

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

by finding an appropriate sequence of legal moves. He develops a rule based expression of the problem solving strategy assuming the Prolog style of program control.

The problem expression is in terms of specialised sorting operations which can perform such operations as:

"put the second row in order, without moving blocks in the first row"

And these are themselves ordered into an effective problem solving sequence. The solution here is algorithmic; Kowalski describes as heuristics a collection of rules which may solve some problems but are not guaranteed to solve all of them. When a collection of rules covers all cases within a class it becomes an algorithm for that class. Rules are held to be an ideal way of developing heuristics because the separation of logic and control components facilitates incremental development.

We can now make a number of observations about the power and limitations of logic programming. Firstly, we note that a persistant theme in this section is that the human program designer must make the problem solving strategy explicit in domain specific terms and cannot rely on an autonomous control regime to turn an arbitrary logical specification into an effective problem solver. We then must criticise existing logic languages (like Prolog) for representing part of the problem strategy

implicitly in their control regimes rather than explicitly. We also note that the structural knowledge (in Clancey's terms) must be made explicit as part of the expression of the problem solving strategy and is not automatically derived from non domain specific principles. Secondly, we observe that there is with this formalism no 'procedural/declarative controversy' - it is recognised that the problem solving procedures (strategies) must be explicitly attended to in the declarative statement of the problem domain. Thirdly, once a problem solving strategy has been enshrined in a particular representation of the problem domain we cannot expect that the general capabilities of the theorem prover will confer more general problem solving capabilities upon the problem representation. This is an important point that needs some further elaboration.

Ignoring for the moment the use of explicit control over the theorem prover, a problem solving strategy is expressed as illustrated above through the choice of subgoals and order in which they shall be pursued. This strategy makes the difference between the theorem prover being able to find a solution on the one hand, or running off into combinational explosions or never ending recursions on the other. Now it is a powerful property of logic programming that rules written to be used in one way can often be used in another by the exploitation of the general proof procedures. Thus, having written a set of rules to determine membership of a set

```
member(X, [X|_]).
member(X, [_|Y]) if member(X,Y).
```

we can also use them to generate members of a given set. The vestige of the declarative/procedural issue is found in the naive view that this

pleasing property of logic programs at a microscopic level extends to the problem solving ability at the macroscopic level. We can now readily understand that we should be very surprised to find that a carefully expressed problem solving strategy for a particular class of problems could solve the inverse class of problems by some simple inversion of the strategy. In any case we may be sure that such a property is not conferred by the simple use of logic to express the strategy. It will only come as a result of deliberate design and insight into the structure of the problem domain. It has not always been adequately recognised that the ability to use collections of logic clauses top down or bottom up does not necessarily extend in any useful way to a general problem solving ability.

Our fourth observation on the use of logic for representation is therefore that the development of the much sought after general control strategies will go hand in hand with the epistemological studies described by Clancey that allow us to tease out problem independent strategies from problem dependent structure. Experience in automatic theorem proving is here a contributor of ideas, but not the only contributor. Analysis of human problem solving in the style of Clancey's work also generates new general strategies.

Finally, we can now understand that logic itself is in no sense a general solution to knowledge representation but rather:

(a) An analytic tool at the knowledge level.

(b) A powerful implementation tool at the symbol level.

We must not let its general power at the symbol level deceive us into believing it brings ready made epistemological solutions to the knowledge level, or that it is the only tool that may be usefully employed for the analysis of that level.

2.7 KNOWLEDGE FOR MACHINES [3]

We conclude this section as knowledge representation by briefly taking up the task that was abandoned at the start - that of reviewing the issues of representation through the techniques that have been used. As a framework for this discussion we take Addis & Johnson [3].

Addis & Johnson first of all develop an approach to knowledge the main points of which may be summarised as follows: There is no intrinsic connection between the signs of a representational language and the world; meaning is only to be found in a shared social context. The representation of knowledge is distinct from the representation of meaning; the process of knowledge elicitation identifies knowledge structures whose representation does not entail representing the meaning of those structures. "Knowing" is having the right to be sure, and the builder of a machine knowledge base must be concerned with transferring the right to be sure which is an essential part of the knowledge context. The possible bases of the right to be sure are briefly examined and the concept of supportive knowledge (in the sense used in Clancey) is introduced. In building a knowledge base a decision must be made on a level at which justification will cease. This level will be determined by the class of users of the knowledge base and will specify base premises assumed to be known. This level will change with time and unless the knowledge base adjusts the knowledge content will degrade. A stronger statement is also made, that knowledge of a concept (not only an inference) is also subject to support through argumentation and can never therefore be considered to be 'in hand'.

Three formalisms for representation are then discussed: semantic nets, production rules and frames. For semantic nets two design criteria are identified:

- the set of tasks to be performed
- the level of assumed knowledge to be expected of the user.

The first of these will guide the choice of organisational primitives and given that there is no universal theory of representations these must be domain dependent. It will also provide the functional specification for the interpreter-plus-network problem solver. The second criterion is concerned with the basis for the support knowledge discussed above. Addis & Johnson note that the particular shortcoming of the net formalism is that it does not specify the interpretive and transformational procedures that may be legally applied. Nets are open to any kind of procedure and this is the source of their power as well as their weakness. The weakness is that an ad hoc procedure is not guaranteed to manipulate all our primitives in ways that maintain real world consistency (unlike logic). The power is that certain inferences of heuristic value may be given precise and efficient expression (see discussion in section 2.3 above).

Production systems allow programs to be constructed from rules of the form:

If <condition> then <action>

Rules may not directly invoke one another but interact through a global database. The selection of rules is determined by a separate control structure. This perspective tends to emphasise the refinement of the problem representation for efficient manipulation. This in turn obscures the problem solving strategy since subgoals are invoked by ensuring that at the correct time their conditions will match the current state of the representation.

Production systems as typically implemented in AI have an essentially weaker control regime than logic languages and therefore make the explicit representation of control (deductive) knowledge and problem knowledge more difficult. This was discussed in detail in Clancey's work which also indicated that the formalism is powerful and effective if used wisely.

A common theme in much of psychology is that we can only perceive in terms of previously established structures; the method by which we acquire such structures posing deep developmental and philosophical problems. The same theme is found in artificial intelligence encapsulated in the frame or script concept. A frame is simply a structure that can be brought to bear upon a situation as a kind of prototype within which the solution can be understood. In a formal sense then it is a technique for hypothesis generation and an expression of the logical constraints between hypothese. Frames are therefore a tool of a different order to the preceding two, being an organisational principle more than implementational device. That said, frame-based languages have been implemented and used effectively. From the perspective of Newell's paper we may assert that the justification for this is a technological issue at the symbol level distinct from any epistemological justification at the knowledge level. At the symbol level they are subject to the same comments as we made for semantic nets.

In the recognition of the importance of user defined units to organise knowledge we return to the problem of determining the basis for the expression of support knowledge. When we realise that humans have the ability to reconstruct these units to meet new situations we realise how far our fixed representations of narrow domains are from conferring true intelligence upon our programs.

2.8 SUMMARY

Given the diversity of views on knowledge representation referred to at the beginning of this chapter it cannot be expected that this survey could reveal any general summarising principles for the field. It does seem reasonable to claim however that Newell's hypothesis of a knowledge level has genuine programatic value in our study of the issues. We can begin to see that arguments about notations have really been about epistemological issues which have come to the fore through the use of those notations. Each notation confers a certain bias on how a worker will express the functional equation that the symbol level is called upon to solve. He will of course express it in a way that suits his notation, but this is a strength as well as a weakness. Every endeavour that enables us to ascribe some degree of rationality to an artificial system has some potential contribution to make to our analysis of the knowledge level.

* * * *

CHAPTER III

INTELLIGENT LEGAL INFORMATION SYSTEMS

3.1 OVERVIEW

The application of the techniques of artificial intelligence to legal systems is still at a comparatively early stage and is characterised by a relatively small number of significant projects with a considerable amount of peripheral activity. The surveys in [27] and [8] reveal an enormous variety of aims, objectives and approaches to the field and only a small minority may be considered to have a background in, and an up to date understanding of, the potential (and limitations) of current AI techniques. It is beyond the scope of this project to survey all this literature which deals with general applications. Instead, following the perception in the last chapter that the application of a particular representational technique contributes to the understanding of the domain, the survey is organised around those techniques.

It is of course the goal of most projects that apply AI to the law to design representations that will serve a wide spectrum of purposes; the nature of the law itself constantly brings this purpose to mind. At any one time it is possible to point to a body of written material and declare that it is the law. Notwithstanding that the concepts embodied in that material are only fully determined within a social context there is no doubt that this one source of knowledge must be common to all the applications of the law. How attractive it is then to believe that we can represent the law just once and only supplement it with additional

world knowledge to prescribe its application to world problems. This may be likened to separating the structural from the strategic knowledge in NEOMYCIN. We saw there that in terms of an explanatory systems this might be a goal, but for a problem solving system it still posed deep problems of the separation of domain dependent knowledge from domain independent application strategies. The law may be considered to pose special problems in this respect since although a purpose guides the framing of a law, that purpose is not contained in the law itself [38]. The conceptual model is specifically absent, and this absence we may assume reflects a judgement by lawyers that it cannot be adequately expressed in legal language. We must therefore expect difficulties if we try to represent that model in the weaker tools we have at our disposal.

Section 2 looks briefly at some of the legal systems that are on the fringes of 'intelligent' systems. In section 3 we look at the LEGOL project which grew out of the traditional systems analysis and database approach, and how a natural synthesis of that notation with logic seems to be suggested. Section 4 then looks at the use of logic for legal systems, which is the primary interest of this study. Section 5 discusses deontic systems - those that concern themselves with the concepts of permission and obligation which are fundamental to any legal system that seeks more than narrow applicability. Lastly, in section 6, we look at frame based systems, particularly the TAXMAN Project and get some feel for the complexity to be expected of a system that can assist the practising lawyer.

3.2 PRECURSORS OF INTELLIGENT LEGAL SYSTEMS

The complexity of the law is such that often even quite modest mechanical assitance with its comprehension or application can yield significant benefit. In [5] a system is reported in detail that analyses tax allowance. The system was written in BASIC and the knowledge is represented entirely in flow charts. A similar system (also in BASIC) is Hellawell's CORPTAX [17] for analysing the taxation of stock redemptions. The weakness of these systems is apprarent: all possible questions that might need answering must be anticipated and catered for. More seriously, the structure of the rules is actually lost in the translation to a flowchart so that modification becomes a programming task rather than a knowledge engineering one.

Selecting welfare benefits is the subject of du Feu's work [11]. This project was motivated by the observed very low rate of take up of benefits and was designed to take a 'whole household' approach. Very few details of the program are given and it appears to use a form of decision table approach.

Gilbert has developed a prototype DHSS benefit assessment system which has been tested in an experiment with Citizen's Advice Bureaux workers in the field. The system is believed not to involve AI techniques but written accounts were not available to the writer.

3.3 LEGOL

The LEGOL project [36, 37, 38] is an ambitious project that has sought to gradually widen the scope of its representational formalism to cover progressively more aspects of legislation. In this section we discuss those aspects of LEGOL that grow out of relational database ideas. The extension to handle deontic concepts is discussed in the next section.

LEGOL is based on a relational algebra for the manipulation of data elements. The semantic model for the defintion of these data elements gives a particular emphasise to the representation of time. All the data elements are recorded with the time at which they begin or end. This is particularly important for legal systems where time often plays an essential role. The data elements or "entities" fall into three classes.

Things - Entities which have an independent existence and whose time period is uniquely determined by the values of the other attributes, eg a person, where the time period represents the life time.

Conditions - These entities have a time period which is not determined by the other attributes.

eg employed (ICI, Bloggs, 1976-1980)
employed (ICI, Bloggs, 1982-1984)

A particular condition cannot be specified independently of its time period since this period is conditional.

States - These are similar to conditions and are differentiated by one attribute being a function of the others, eg number_of_children (Family, N, period), N is a function of Family and period.

However function is not a defined semantic concept and the distinction between conditions and states appears to be more intuitive than formal.

We may see in this model some similarties with the conceptual model for relational databases expounded by Addis [2], but a full comparison is beyond the scope of this project. The LEGOL language is a means of manipulating an underlying database in which all these entities are held in the form of relations. The process of representing the law with LEGOL therefore consists in two phases. First the entities to be represented must be determined through a process of relational analysis; secondly the legal rules must be expressed in LEGOL rules for the manipulation of the relations. We can mention the hope expressed in Stamper [37] that it will be possible to reach a 'canonical' analysis, a goal of relational analysis in general; Addis [2] shows how any such analysis is incomplete unless the world constraints between relations have also been represented. This does not seem to have been tackled by Stamper except in the informal condition/state distinction.

The syntactic unit in LEGOL is the rule, and its form is:

<target relation> <update symbol> <source expression>

The source expression comprises relations and LEGOL operators. The effect of the rule is to assign the result of evaluating the source expression to the target relation. The operators are specialised forms of the relational operators to take account of the special importance given to the time attributes. The following example is taken fairly directly from [37]:

number_of_children(Jones,3,1959-1967)
number_of_children(Jones,2,1967-1972)

These represent the number of children in the Jones family in the given periods (where defintion of a child is held in rules elsewhere). Family Allowance rates are held in another relation:

rate(2,8/-,1965-1970)
rate(3,18/-,1965-1970)
rate(2,25/-,1970-1979)
rate(3,40/-,1970-1979)

Given these relations a Family Allowance rule can now be written:

allowance(Family,X)← rate(N,X) while
number_of_children(Family,X)

This rule performs a join over the number of children with special handling of the time attributes. The 'while' operator creates for each tuple in the target relation the time attributes that represent the intersection of the periods in the tuples of the source relations. Where there is no intersection the tuple is deleted.

The effect of the allowance rule on our example data is thus to construct the new allowance relation:

allowance	Jones	18/-	1965-1967
	Jones	8/-	1967-1970
	Jones	25/-	1970-1972

LEGOL has other special time operations such as 'or while', 'whenever', etc. These are all defined within the interpreter and not accessible to argumentation by the LEGOL user.

Two update symbols are provided in LEGOL, represented by a single arrow (as above) or a double arrow. The double arrow form transfers the candidate key of the evaluated source to the target, whereas the single arrow form does not. It is also possible to name attributes of the target relation and update them by some operation on the candidate key. For instance, the rules:

```
start_of child(Person) ←start_of Person
end_of child(Person) ←start_of Person + 16
```

together model the rule that a person under 16 is a child.

Only the simplest legal rules will be represented by a single LEGOL rule. The interpreter supports multiple rules through sequential interpretation and it has also been found necessary to introduce iterative loops, and the other control structures of conventional programming languages to deal with the complexities of real law. This is a serious shortcoming and indicates that, quite apart from any conceptual problems there might be with using the relational model as the basic semantic framework, that the operators and their interpreter are fundamentally inadequate for representing the law. The criticisms that applied to the BASIC programs described in section 2, with their confusion of program structure with legal structure are seen to be still present in the LEGOL formalisation.

In [32] Sergot performs a thorough comparison of LEGOL with logic programming and shows not only that all its features can be readily reconstructed in logic, but that considerably more power resides in the new formulation without being attended by any obvious penalties. Taking the family allowance example we can represent it with corresponding logic clause:

```
allowance(Family,X,T) if rate(N,X,T1),
                         number_of_children(Family,NT2),
                         while(T1,T2,T).
```

Here the time attributes and the relationship between them in the source and target has become explicit: while(T1,T2,T) will hold whenever the intersection of time periods T1 and T2 is the period T.

When run bottom up this clause will have the same effect as our LEGOL rule, adding new allowance clauses by inference on the supplied data.

However, it can also be used top down to establish a particular instance of an allowance from a database of families and rates. The benefit of such flexibility is obvious, and becomes even greater when disjunctive rules are considered. A disjunctive rule is one of the form:

A if B or C or . . .

To establish A it is sufficient to establish only one of B,C . . . A logic program using A top down is able to do this. A LEGOL program must compute B,C . . . and then their union. This may be not only inefficient, but could in some instances be combinationally explosive and non terminating. Given that disjunctive conditions are a common feature of the law this is not a trivial problem.

We therefore turn to systems that have used logic directly for legal systems and discuss their design and current capabilities.

3.4 LOGIC BASED SYSTEMS

As we have seen from preceding parts of this study there are adequate reasons deriving from AI experience for supposing that logic could provide a suitable representation for some aspects of legal systems. Independently of these considerations however there is also the evidence from the legal profession that logic is relevant as a means of structuring law. In [4] Allen discusses in detail the structure of written legislation and concludes that while legal drafters are skilled in handling the semantic dimensions of the written word they have totally

failed to handle structure in any consistent or systematic way. The result of this failure is inadvertant ambiguity due to the multiple interpretations that can be put upon collections of statements. Computer science long ago confronted the ambiguity of such constructs as the 'dangling else' and established rules to deal with them. Legal drafters it seems have never approached their task with the same rigour. Since the drafters have at their disposal a considerably richer language than the computer programmer the consequences are correspondingly worse. In his carefully argued paper Allen accounts for over 4000 possible interpretations of a set of statements whose connectives include the word 'unless'. This complexity arises only when the statements include the deontic concepts, but whatever the reason it is plain that the situation is unsatifactory. The solution proposed by Allen is a move towards "normalised" drafting in which only a standard set of structuring primitives would be used. Not surprisingly the structures he proposes to express the relationships between normalised statements are the logical ones: conjunction, disjunction, conditional, etc. Normalised statements must all contain deontic operators expressed in a standardised way.

Allen's study sounds a note of caution when we set out to represent legislation. The law is what is currently written down and that takes no account of Allen's proposals. We shall have to choose in our systems whether we translate the law in normalised form first (current practice) or whether we attempt to model its ambiguity. We should not suppose however that the ambiguity makes legislation unamenable to logic. Logic is quite at home with ambiguity and indeed is the tool with which Allen teased out all the meanings of 'unless' - it is unlikely that anything other than logic could adequately represent them.

Now that the use of Prolog as a logic programming tool is becoming widespread a number of informal reports are made of projects, on representing rules and regulations. The only documented case study available however is the work of Hammond [14] on DHSS supplementary benefit regulations. More recent work at Imperial College has used logic to represent sections of the British Nationality Act and the author is grateful to members of the College for access to informal documentation of this project.

Hammond's project started with the clearly defined aim of expressing entitlement to supplementary benefit. With this target the source material was not the legislation itself but the pragmatic expression of it in the DHSS guide to the application of the legislation. The rules were elaborated from this guide with the aid of a DHSS expert. The final description of benefit entitlement contains a little over 200 rules and facts and includes computation of the entitlement. It is important to note that the project stated with a single goal, which in effect translated into a single top level goal for the Prolog program:

```
Person is_entitled to sup_ben if
  not Person is_disqualified_by sex and
  not Person is_a_juvenile and
  Person study_status_OK and
  Person is_a_GB_resident and
  Person is_excused_or_registered_for_work and
  Person needs_financial_help and
  not Person is-disqualified_by trade_dispute.
```

Most of these conditions are themselves described by further rules. It is the goal structure of the program that determines how the rules are structured, not any underlying conceptual model of the 'benefit domain'. At some point in pursuing subgoals the program must either encounter assertions or fail in its search. The logic system used (Query-the-User [32]) allows condition to be declared "askable" and this has the effect that the user of the system is asked to supply the information to satisfy the goal. This has its most obvious use in acquiring basic data such as 'age', but can also be used to allow an external decision on something that is essentially undecidable within the knowledge available to the system, eg discretionary judgements.

The system is able to offer an explanation of its reasoning by reciting the trace of successful goals; in this respect the program does nothing not done equally well by non-logic systems.

The ability to make flexible access to the information in the program is quoted as a benefit of the logic representation, eg:

> "What is the maximum capital allowance for supplementary benefit claims"

or

> "What is jones disqualifed by"

These queries must be put to the program in a standardised form, ie

> Which (X : jones is_disqualified_by X)

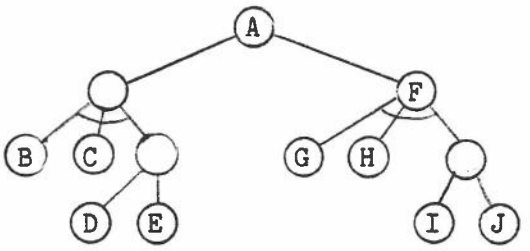
The symmetry of the logical representation thus allows a relation such as 'is_disqualified_by' to find a solution as well as showing that a

relationship holds. In [21] the distinction between finding and showing is discussed. Since any procedure that applies to showing problem p(T) also applies to identifying an individual X such that p(X) it follows that the search space for a finding problem is generally larger than that for a showing problem. Following our earlier discussions of the need to include problem specific information to guide the search strategy we must recognise therefore that the strategy that works for showing will not necessarily work for finding. The Hammond program which has a fairly simple structure amenable to exhaustive searching does not reveal this problem, examples which do are given in the next chapter of this study.

We also note that the order in which questions are asked of the user is entirely determined by the order of search. While this is perhaps satisfactory in a simple case we would probably like to have more control in the general case and it would be advisable if dialogue control were explicit. In practical circumstances carefully ordered questions are asked by benefit assessment officers of claimants. It should be possible to specify this order and maintain it if some changes to the legislation require modifications to the problem solving strategy. In this example the problem solving strategy would appear to generate a sensible dialogue but we shall see that this need not always be the case. These remarks are not to be taken as criticisms of the representations in logic: in his study of MYCIN Clancey notes that there is no means of controlling the ordering of rules (and hence dialogue) that are selected to try to establish a goal. The ordering is determined by the order in which goals were edited into the system. Prolog is superior in this respect in that goal ordering is explicitly controlled. We simply have to be careful not to expect too much of the general deductive power of our logic system.

Another example of a Prolog program encoding legislation is the system under development at Imperial College by Sergot et al to handle the British Nationality Act. This is very similar in style to the Hammond example, using Query-the-User to couple the system user into the problem solving process. Unlike the supplementary benefit program however it was designed directly from the legislation. This particular legislation has been very amenable to logical representation since it is logically straight-forward and self-contained. Like the Hammond program it falls into a simple goal structure which can be adequately represented by and-or trees. The following example, shown in Figure 1, is taken from informal project documentation.

This can be translated directly in Horn clauses in a Prolog program by the translation of the tree:



to the clauses

A if B and C and D

A if B and C and E

A if F

F if G and H and I

F if G and H and J

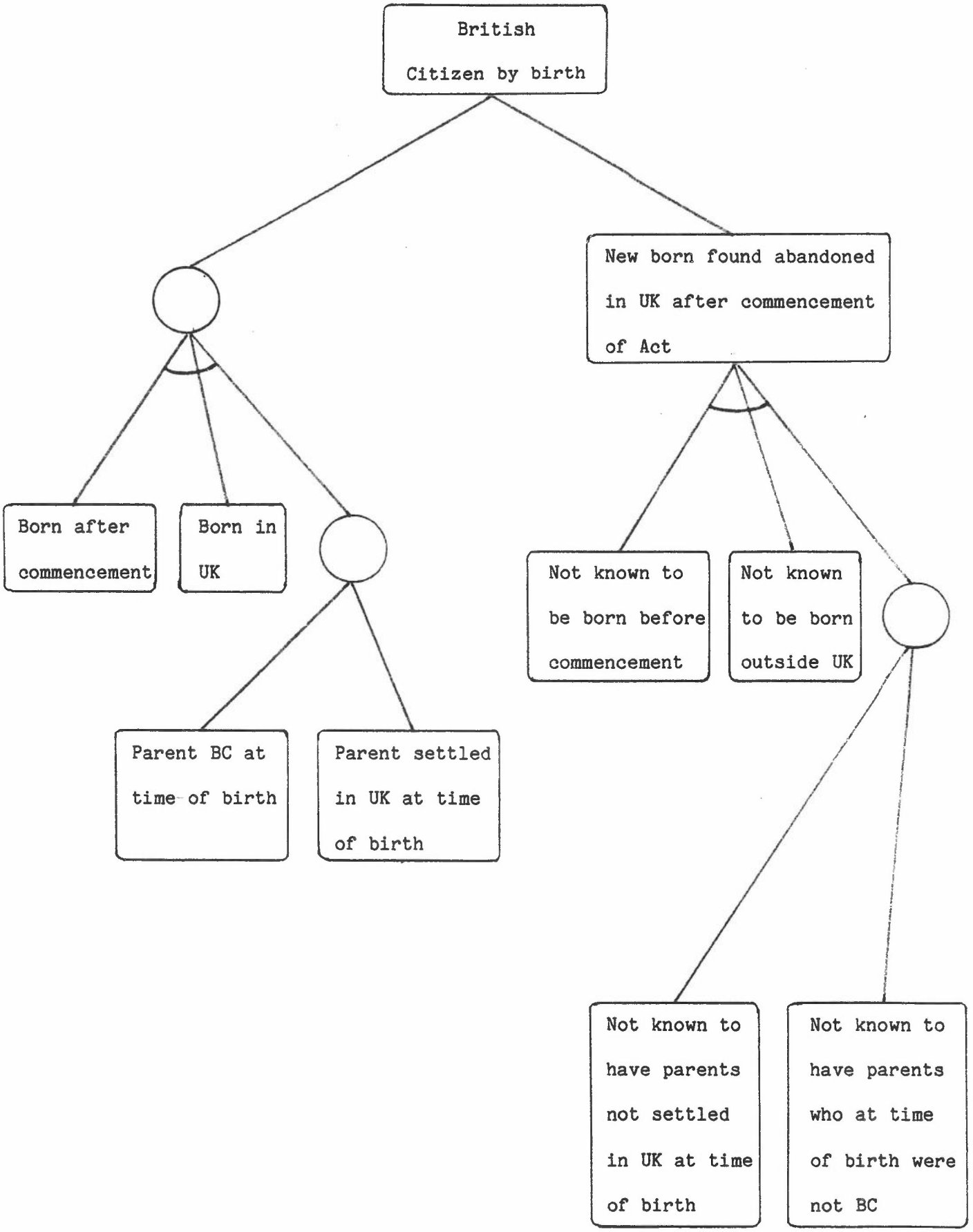

British
Citizen by birth
New born found abandoned in UK after commencement of Act
Born after commencement
Born in UK
Not known to be born before commencement
Not known to be born outside UK
Parent BC at time of birth
Parent settled in UK at time of birth
Not known to have parents not settled in UK at time of birth
Not known to have parents who at time of birth were not BC

Figure 1

In [21] Kowalski shows how in general the and-or tree representation of a problem does not make explicit the effect of subgoal selection strategy on the size (and finiteness) of the search space. An extended representation is necessary to show the contribution each procedure makes to the values of the variables to which the procedure is applied. Like the supplementary benefit example the British Nationality Act program presents a sufficiently constrained search space that this is not a problem.

We can summarise the experience gained from these systems briefly as follows:

(1) A logic formalism allows a natural expression of rules in a form that is easily seen to correspond to the legislation.

(2) The rules are easy to modify to reflect changes in the legislation.

(3) Surface level explanation of reasoning is easily provided. Since it is the rules that are the law, and not any underlying deep concepts, issues of deeper explanation do not apply. However, we consider the limits of this understanding of consultation systems further in section 5 below.

(4) These systems have not had to tackle any serious distinction between expressing the law and applying it to solve problems. In the Hammond example the program was designed to solve a problem, and its rules are not constrained to be identical to those in the

legislation. In the BNA the direct representation of the legislation appears to have resulted in a useful problem solving system. The extent to which this approach may be extended is an open question and one of the subjects of the next chapter.

3.5 DEONTIC SYSTEMS

To a first approximation the systems described so far in this chapter have been concerned with handling objects and realationships rather than the deontic concepts of permission and obligation. These concepts however pervade the law and in some sense may be regarded as its essence.

In the proposals of Allen [4] for a normative form for legal drafting discussed above it is required that the consequent of every rule contained one of the deontic concepts. These concepts effectively express the relationship of individuals to the application of the law by defining legal and illegal behaviour. As our representations of the law are extended to cover sequences of behaviour we shall have to tackle the definition of these deontic concepts. It is not proposed to study them here since that is the province of lawyers (see eg [25]) and it is a subject that fills many volumes. Here we simply note the type of problem that has to be tackled and the two stances that can be taken to the use of logic in automated systems.

Given legal rules of the form:

X is obliged to do A for Y

Y is permitted to not do B

We would like to be able to deduce the inverse relationships that would allow us to answer such queries as:

Has Y a right to A?

Is Y obliged to do B?

Where the actions of agents produce consequences that appear in further deontic rules we can find considerable complexity in the relationships involved (as illustrated by the 4000 meanings of 'unless'). Two attitudes can be taken to logic in the representation of these deontic concepts and the rules of inference that a study of legal processes deems to be appropriate for their manipulation. These two attitudes are the same as those found in AI to the treatment of uncertain or probabalistic reasoning. In MYCIN and many other systems it is possible to attach a 'certainty factor' to a rule, eg

if P then Q (C1)

if Q then R (C2)

The transitive certainty factor C3 for the rule

if P then R (C3)

is then defined implictly within the system by some ad hoc definition such as C3 is the minimum of C1 and C2. In this approach therefore the underlying deductive mechanism is altered to cope with a perceived mismatch between the properties of the domain and two-valued logic. The alternative approach (adopted in [10]) is to represent the special inference rules explicitly. Our example could then be expressed:

```
leads_to (X,Y,C) if causes (X,Y,C).
leads_to (X,Y,C3) if
        causes (X,Z,C1) and
        leads_to (Z,Y,C2) and
        combine (C1,C2,C3).
```

The combine relation can then be defined as appropriate, and obviously alternative defintions could be given for different relations by simple extension. Not only does this approach make the inferential rules explicit and therefore amenable to explanation, modification, relation-specific definition, etc, it also leaves us with an underlying deductive mechanism whose theorem proving properties are well understood. An ad hoc logic designed to fit our special requirements of some particular reasoning system is unlikely to possess the same properties.

3.6 FRAME SYSTEMS

The TAXMAN project of McCarty [23, 24] is a major attempt to use artificial intelligence techniques to model legal reasoning and has probably gone further in analysing the conceptual problems than the other system we have described. The LEGOL system, which is the only other project to try and define a formal representation has specifically excluded from its goals automatic legal reasoning.

TAXMAN models one sub chapter of the US Internal Revenue Code - the taxation of corporate re-organisations. The legislation is very complex and has been the subject of a number of judicial decisions so a complete model must account for both statue and case law. The goals of the project are ambitious, aiming to take the automatic modelling into the realm where the concepts as well as the inferences must be supported by argumentation.

The model developed by McCarty to support his system goes beyond a surface representation of the rules and is expressed in a network formation as an abstraction/expansion hierarchy of frames or templates. Rather than show the notation used in [23] for the semantic networks we reconstruct them here in the notation of partitioned networks defined by Hendrix in [18]. There are insufficient details in the TAXMAN papers available to the writer to be sure that this reconstruction captures all of the original but it serves its main purpose. That purpose is to show that although the original work is expressed in a semantic network and implemented in a frame based language (AIMDS) it can equally well be expressed in a semantic network whose equivalence to predicate logic has been established. In [18] Hendrix establishes this equivalence, showing further how his notation is a means for representing logical statements about collections of propositions.

Figure 2 therefore is an expression of the basic concepts of the TAXMAN model in the Hendrix notation. We note that there is no reference in McCarty's work to Hendrix or others who have used the semantics of predicate logic for their network formations and his choice of frames was not necessarily therefore fully informed.

In the figure the table on the arrows have the following meanings:

s	set inclusion
ds	disjoint subset
e	set membership

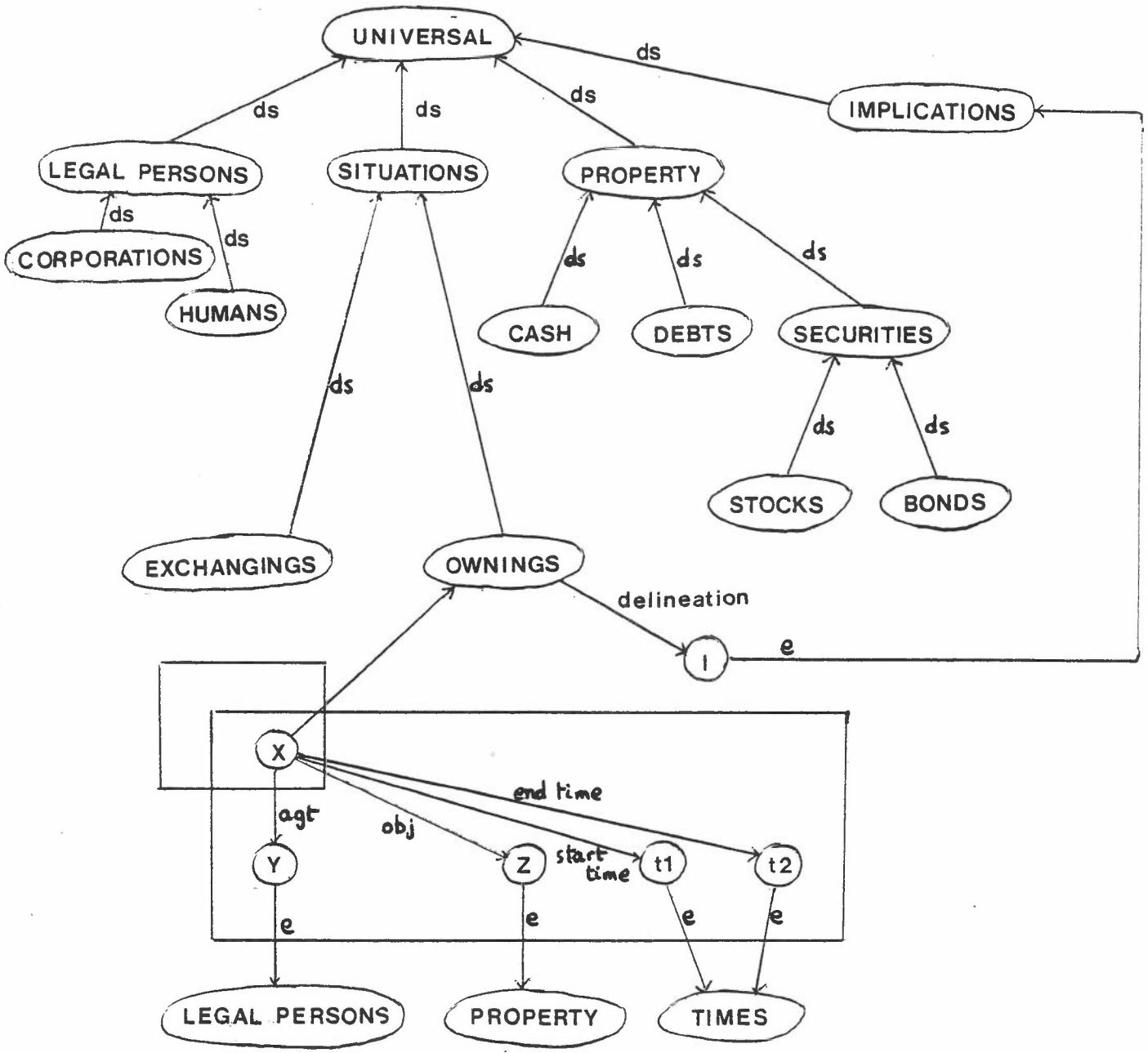


Figure 2

Figure 2 shows not only the hirarchial organisation of concepts but an example of a template; that for the 'ownings' mode. In the Hendrix notation every situation set has a template, or delineation in his terminology. A delineation specifies the deep cases that name and restrict the participants of situations in the set. The ownings template delineation shown corresponds to the formula:

$$\forall x \{ \text{member}(X, \text{Ownings}) \Rightarrow \exists y, z, t1, t2 [\text{member}(y, \text{Legal persons}) \& \text{agt}(x,y) \& \text{member}(z, \text{property}) \& \text{obj}(x,z) \& \text{member}(t1, \text{Times}) \& \text{Start\_time}(x , t1) \& \text{member}(t2, \text{Times}) \& \text{end\_time}(x, t2)]\}$$

This is a formal statement that all the named slots must be filled in to create an instance of an ownings situation. Other templates are defined by McCarty but their definition would add nothing to this description. McCarty notes that this model can be used both top down to find instances of some concept or bottom up to establish membership of some class. Clearly the frame language is no different to the Hendrix system in this respect. Where the AIMDS system appears to be useful in the TAXMAN system is in its ability to return partial matches to a concept together with a residue expression which lists that part of the logical expression and its associated binding list which produced the true, false, or unknown evaluation, respectively. The system used to implement the Hammond programm [15], also has an ability to record dependencies within a proof on undertermined variables, but the writer has not sufficient details to make a detailed comparison.

McCarty is interested not just in identifying states but in reasoning about the state changes involved in company reorganisations. There we have the Exchangings subset of Situations with slots for "agents", "object", "old owner", "new owner", "instant time". These are the rudiments of the TAXMAN I system and can be used to express the essential aspects of its domain. McCarty however identifies some severe shortcomings of the system that have lead him to investigate the design of a TAXMAN II system. First, the system must represent at the ground level the full set of factual situations that might occur in any given case. McCarty concludes that it is inconceivable that a full set of facts could be expressed in the TAXMAN I formalism in sufficient detail for the system to be capable of dealing with any interesting legal question. Secondly, McCarty cites the "open textured" nature of the high level concepts involved in legal reasoning as being essentially beyond the scope of the formalism. The legal concepts of importance he identifies as "dynamic" rather than "static", and justified by a sense of "purpose", a point that we have already seen as a potential problem in other domains.

To tackle these problems McCarty proposes a model which goes beyond the fixed-template/partial match capability of TAXMAN I and instead has concepts modelled as a "prototype" and a sequence of "deformations" of the prototype. A prototype is a concrete description and deformations are mappings of these descriptions. A concept then is the set of examples which can be generated by a sequence of mappings from the prototype. The theory is not worked out in detail in [23] where McCarty concentrates on a mixture of theoretical and implementational problems.

The most interesting aspect is that he finds a theory of the deontic concepts to be essential to his purpose. This is because the domain is intimately concerned with the maintenance or transfer of rights and so a particular pattern of events can only be judged to fit a certain concept if it maintains complex patterns of rights in property.

Seen as Situations in the Hendrix scheme the new conepts do not appear to pose any new representational problems. The problems clearly lie in the semantic modelling of the domain. In summary therefore we can conclude that the frame based approach helped McCarty to handle the analysis of his problem by suggesting an organisation in terms of templates, inheritance hierarchies, collections of propositions, etc. However, the particular choice of frame language may have made his task more difficult by giving him only a subset of the power available in a notation such as Hendrix's which has full logical adequacy.

* * * *

CHAPTER IV

REPRESENTATION OF SSP LEGISLATION

4.1 INTRODUCTION

This chapter reports on a practical investigation that was undertaken by the writer into the representation of legislation using PROLOG as a logic programming language. The piece of legislation chosen was the Statutory Sick Pay (SSP) provisions contained in the Social Security and Housing Benefits Act 1982 [35] and associated Regulations [39]. This was chosen because of the widespread attention it received in the computer press at the time of its introduction. Since it concerned pay-roll programs of DP departments its provisions had to be incorporated into those programs, which are written in traditional DP languages (usually COBOL). The legislation was strongly criticised by the DP community because of its complexity; it was said to be both difficult to understand and difficult to implement. This legislation therefore suggested itself as a suitable test-bed for the technqiues of knowledge based programming for which much is claimed for their ability to tackle complex problems. The choice of logic programming language was also quite natural. Regulations are expressed as rules, and indeed we have seen in our survey in the previous chapter that a more explicit and carefully structured representation of rules has been proposed for the law. Logic is a natural and powerful expressive tool for representing rules and would therefore seem to be an obvious choice. PROLOG was the only practical system available, and

there is some interest in finding its limitations as a logic programming language; but that was a secondary objective to exploring how readily the logic rule approach could generate a system of useful capability within the time constraints of the project.

The project was concerned not only with using the rule based approach to legislation but also with testing how readily the logic rules could be derived directly from the written legislation and how close they could remain to it while acquiring some useful problem solving ability. From one point of view one could argue that since the written law is not tailored to any one of its many domains and methods of application nor should its machine representation be. That this naive representation is unlikely to have much problem solving power we have seen from earlier chapters. The project deliberately started with this naive position and went through the following stages:

(1) Direct representation of rules as close to their written form as possible with no particular problem task in mind. Identification of limitations.

(2) Representation of rules in a form tailored to a specific top level problem goal in the style of the Hammond Supplementary Benefit program, but still keeping as close to the written form of the legislation as possible.

(3) Representation of the legislation in a form powerful enough to tackle a number of real world case histories. The example problems were taken from the Employers' guide to SSP produced as an explanatory document by the DHSS.

In the course of this evolution the general target was a system that could be consulted in a fairly general and 'intelligent' way about the provisions of the legislation. As the project progressed it became apparent that to build such a system four categories of information would need to be explicitly handled. The inclusion of each type of information entailed going beyond the straightforward translation of the written legislative rules into logic rules, demonstrating that the natural match between the domain and the formalism was superficial only.

Firstly, the law is in general definitional rather than algorithmic in character. The definition of a concept is given rather than a method of discovering or establishing an instance of that concept. We may compare it with a definition of sortedness:

```
sequence y is a sorted version of sequence x if
        y is a permutation of x and
        y is ordered.
```

We could represent this in logic as

```
sort(x, y) if
        permutation(x, y) and
        ordered(y)
```

As pointed out by Kowalski [22] this is more like a specification than a program. To turn it into a program we would wish to transform the definition into an algorithmically more powerful one by the inclusion of specific sorting expertise. In the case of the law we are interested in preserving the original definition, alongside any problem solving strategic knowledge we have to bring to bear. In the British Nationality

Act program it appears that the additional knowledge can be added as additional rules rather than as a modification of the definitional rules. Such a neat distinction proved impossible to maintain for SSP. The relationship between the problem solving knowledge and the definitional law is similar to that between the 'heuristic' and 'causal' rules in Clancey's analysis. The law is supposed to be a set of rules to which the behaviour of society will conform, it is a model against which a particular case should be made to fit. The description of the model however is not the best heuristic for diagnosing the peculiar characteristics of a specific case.

Secondly, to support our problem solving rules we need to establish some underlying conceptual framework for them to handle. This was evidently true in the TAXMAN program which had to deal with the ill structured nature of case law and is no less true when we have only statue law to consider. We cannot therefore assume that because the law is written down that there is no knowledge acquisition problem in representing it. This conceptual framework will have to encompass both common world concepts, of time periods, events, etc, and as our system becomes more ambitious the social context which gives the law its validity as a set of 'norms'. Studying this requirement will help us understand the way we might design a system that can answer questions both about the law, and about its application to cases. Questions of the first type are:

"What is the law on X?"

"How does pregnancy affect entitlement to sick pay?"

"What can I do if ...?"

Questions of the second type are:

"What does the law prescribe about X in this case?"

"What SSP is due for 12 May given ..."

Thirdly, the written law, considered as a specification, is itself an object that presents problems in its representation. The method of presentation of the law is frequently to state a general principle followed by a number of qualifications and exceptions. Where these are presented in close sequence they often only differ syntactically from the formalised rule notation proposed by Allen. Often however the cross referencing is at a conceptual level and modifies the meaning of concepts. Rules can assign special meanings for other rules in particular contexts, suspend them altogether, or regulate their effect in a number of ways. There is no locality of reference to basic concepts and the cross referencing between rules is not always explicit. These problems bear a strong similarity to those studied under the heading of non-monotonic logic. In monotonic logic the addition of new axioms to a system can only increase the number of theorems that can be proved. In non-monotonic logic the addition of a new axiom can render a previously proved theorem false. This problem has not been tackled at all in this project since non-monotonic logics are themselves a research area, but it may be seen as an interesting future area of study. What we can note here is that in some way the written form of the law must be explicitly handled and kept intact in any large scale system if there is to be any hope of keeping in step with the law as it changes.

The fourth type of knowledge to emerge as requiring separate and explicit treatment was the knowledge required to produce sensible dialogues during a consultation. We have seen how the previously discussed examples used the 'askable' label to indicate that the information could be asked of the user when required. This 'call by need' approach was found to be inadequate in its simple form. At the minimum it appears to be sensible to be able to generalise a specific question, eg instead of asking:

"Was <person> sick on 12 May?"

we would ask

"How long was <person> sick for after 12 May?"

Specific problems that arose are discussed later with a sketch of a solution in this case. A consultation system that was to be used by a benefit assessment officer in claimant interviews would certainly have to have a solution to this problem. The DHSS produces handbooks for its officers giving them guidance on the order in which questions should be asked. These rules often go quite outside the benefit under consideration, eg when registering for supplementary benefit a claimant is told to register for unemployment benefit first; fulfilling this requirement automatically establishes data relevant to the SB claim.

This project has concentrated on the practical investigation of the first two of the above four types of knowledge, with some observations on the fourth. The remainder of the chapter describes the project according to the three stages described above. A few caveats are in order before the

description. Firstly, the purpose of the project was to gain insight into the problems rather than to produce a complete polished system for other users. To this end the analysis was taken to a point where problems of complexity became apparent, but were not pushed into elaboration of detail that added nothing new. This said, the final stage was taken to the point of being able to solve real problems in order to obtain some measure of the investment of effort required by the technique. Secondly, as will become clear, only a small fragment of the total SSP legislation has been tackled. The information necessary to answer most of the example queries is contained in 7 sections and a Schedule of the Act and 7 sections of the Regulations. The relevant part of the Act contains 26 sections and Schedules running to several pages; the Regulations contain 22 sections. These other sections range over wide areas, such as records to be maintained by employers, determination of disputes, relationship of such terms as "earnings" and "benefits" to other legislation, etc. In general these other sections possess much less logical structure than the main part concerned with defining the essential conditions of entitlement. To tackle them would raise the problems of the four types of knowledge by an order of complexity and would certainly also entail tackling representation of the deontic concepts.

4.2 THE "DIRECT" APPROACH

In order to qualify as a day for which SSP is due a day must meet three basic conditions and must not be excluded by reason of any of a number of supplementary conditions; the structure of the legislation expressed in the first three sections of the Act is as follows:

employee_is_entitled_to_SSP_for(Day) if

part_of_a_period_of_incapacity_for_work(Day) and

within_a_period_of entitlement(Day) and

is_a_qualifying_day(Day) and

not is_excluded_from_SSP(Day).

We will consider in more detail the first of these three condtions. Two rules contain the essential definition of a period of incapacity for work (piw).

2 (2) In this Part "period of incapacity for work" means any period of four or more consecutive days, each of which is a day of incapacity for work ..."

2 (3) Any two periods of incapacity for work which are separated by a period of not more than two weeks shall be treated as a single period of incapacity for work.

Between them these two rules will demonstrate many of the problems we encounter in the translation into a representation for an intelligent consultation system. The statement of the first rule in the standard form of logic is:

$$\forall(x)[F(x)\&\forall y[W(x,y) \rightarrow S(y)]] \rightarrow P(x)$$

where

F(x) means x is a period of four or more days

W(x,y) means y is a day in the period x

S(y) means y is a day of sickness

P(x) means x is a period of incapacity for work

We can transform this by routine procedures into clausal form:

$P(x) \leftarrow F(x), S(d(x))$.

$P(x), W(x,d(x)) \leftarrow F(x)$.

In the transformation we have had to introduce the function d(x) in order to eliminate an existential quantifier. We have also ended up with two rules, instead of one, the second of which is not a Horn clause. It is plain that we cannot render these clauses back into an intelligible English form that still is an obvious expression of the original rule. Nor are we able to use PROLOG as a problem solver since it is restricted to the Horn clause subset of clausal form. It is possible to derive a corresponding Horn clause specification but only by taking a particular representation, eg lists, of the concept we are trying to define and defining it as a recursive procedure. This takes us into the issues tackled in the later stages of selecting representations and procedures to search them. Before leaving this rule we also note that taken together our top level goal and this one could not tell us whether a day was in a period of incapacity for work. A link is missing that must be expressed by the rule:

```
part_of_a_period_of_incapacity_for_work(Day) if
        period_of_incapacity_for_work(P),
        part_of_period_(P,Day).
```

This is a trivial example of how we must insert 'problem solving' rules in intimate relations with our legislative rules if they are to have any pragmatic value. Further, we realise on reflection that in a pragmatic sense the rule does not say what it means. It is quite clear from the use made of the definition that a period of incapacity for work is not just a consecutive period of sickness but the longest such period. It begins when someone falls sick and ends when they are better; a subset of a period is not properly speaking a period in the sense meant there. To express this we must start to add still more rules whose form is far from simple.

The second of the above rules causes us even more trouble. First of all, this is an excellent example of a rule conceptually modifying one that has gone before to such an extent that the first rule is almost useless. We must therefore throw away our direct representation of the first rule and take the two together; the writer can think of no other way of dealing with "shall be treated as" other than defining what a piw is. An appropriate definition might be:

> Periods of four or more consecutive days of sickness, separated by not more than two weeks, together comprise a piw.

Trying to express this in logic the following suggests itself:

$$\forall x \forall y \forall z [[W(x,y) \& W(x,z)] \rightarrow [FS(y) \& FS(z) \& \neg G(y,z)]] \rightarrow P(x)$$

where

W(x,y) means y is a day in the period x

FS(y) means y is one of four or more days of sickness

G(y,z) means y and z are separated by more than 14 days of non sickness

Clearly, FS and G need further expansion, and this formulation suggests that we would have been better off starting with a definition of a part-piw and then defining how parts comprise a whole. This approach is taken in our final representation. All remarks made above for the first rule concerning the difficulty of expression in Horn clauses and making contact with problem solving rules apply with even greater strength to our now considerably more complex definition.

The conclusion of this analysis is not that logic cannot be used to represent the law. It is that a "direct" approach, unmotivated by a conceptual representation of our task or problem domain does not yield a representation of any pragmatic value, either as a definitional structure or a problem solving tool. Further, that the direct representation may fall beyond the scope of our Horn clause problem solving mechanisms. The translation to logic was comparatively simple and so the claim for logic of a certain 'naturalness' as a specification language in this doman may be considered to be substantiated. We now see how far beyond the specification of the law we must go to specify a system that can apply it.

4.3 THE "SINGLE GOAL" APPROACH

We have seen how to give some pragmatic value to our representation we must constrain it to the Horn clause form and must supply missing information to link the basic rules together. The second approach to the problem was therefore to take the same initial clause as before and treat

it as the top level goal of a problem solving representation. The process of representation is then driven by the top down refinement of subgoals, and legislation is only included if it is encompassed by this process. Appendix A shows the program that results. We find that the two rules we considered above enter into our representations in quite a different way. First of all we have:

```
part_of_a_period_of_incapacity_for_work(Day) if
        day_of_sickness(Day), and
        one_of_four_or_more-days_of_sickness(Day).
```

The first subgoal can then be further refined according to the detailed provisions in the Regulations for determining days deemed to be days of sickness, eg

```
day_of_sickness(Day) if
        under_medical_care_in_respect_of_disease_or_disablement(Day) and
        employee_has_done_no_work_under_the_contract_of_service(Day).
```

The second subgoal contains the essence of the first of the piw rules. No further refinement can be made without making some further assumptions about the problem solving task. The simplest assumption is that this subgoal will be resolved by the user and we can therefore declare this relation to be 'askable'. To determine it with respect to some database representing a case history would require more rules to examine adjacent days and count them.

Although the above encoding may appear quite trivial it is the result of

a design decision that had to be reached independently of the legislation. A more straightforward rule, closer to the legislation, would have been:

```
part_of_a_piw(Day) if
        one_of_four_or_more_days_of_Sickness(Day).
```

By including the additional subgoal it is possible to bring in the full definition of a day of sickness (which comes down to five bottom level goals to be declared 'askable'), before asking the generalising question. In this way the user is lead through the detailed definition for one day and then asked whether it belongs to a longer period of such days. We have not by these simple rules avoided the complication of handling the definition of a piw as distinct from deciding whether a day is a part of one: the determination of other goals reached through our top down refinement demands that we discover when the piw started to which this day belongs. In this case it was not possible to keep the definitional rules separate from the heuristic ones as it was for the rules establishing that a day was in a piw.

The bottom level goals that result thus have a rather contrived air about them:

```
beginning_of_period_of_sickness_including_day(Day,Start)

start_of_period_of_sickness_greater_than_four_days_ending_
        within_previous_fourteen_days(Day,Start)
```

Alternatives of more or less elegance could be produced but the points are well made that problem solving rules are very different from those derived directly from the legislation, and the retrieval of definitions is a different task from applying the definition to a case history.

Further examination of the rules in this version of the program shows that many of the relations are highly dependent on their context of use for their meaning and value. Specifically, where several subgoals of one rule are all 'askable' the later ones assume the dialogue context created by the previous ones. In most cases this could be avoided if desired by reducing the subgoals to a single more complex one, or by passing variables between them, but that merely makes the 'askable' goals more difficult for the user to deal with.

The consequence of the two preceding observations is the conclusion that this representation will be highly inflexible. The large number of bottom level goals is shown in the Appendix and the highly specific nature of many of them shows that they are unsuitable for a general problem solver able to reason bottom up from data. Because of this the actual program written did not parameterise the rules on the day under examination. It was quite evident that the program could only possibly show that a day satisfied some conditions, not find those that did.

It was the intention when this project started to obtain a version of Query-the-User that would allow a program written in this way to be tested. Unfortunately this was not possible and so the documentation in the Appendix represents a design rather than a proven program. It was not therefore possible to give examples of its operation or explore its properties through example problems.

In certain places the refinement process has been stopped short and this is indicated by ???. At these points trying to fit the representation into this approach became very convoluted and was abandoned.

The Query-the-User method has proved very successful in other instances so we must explain why we had problems with it here. The answer is clearly that the system's internal reasoning powers are far too weak. It is driven back to the user for help with evey step along the way. In fact the program is very little more than a decision tree. It lacks any model for achieving coherence between the many subgoals to be determined. Asking the user is all right if there is a good match between his real world concepts and the goals that the system must satisfy; in order to achieve that we must turn to the third stage of the project.

4.4 THE "CONCEPTUAL" APPROACH

At this point it was clear that representing the legislation for explanatory purposes and problem solving purposes are two distinct tasks. Sometimes they can sit uneasily together, sometimes they are in conflict. For this stage the original aim of producing a problem solving system for the examples in the employers' handbook was adhered to. A note on extending the system to the first task is included at the end of the section. An attempt was still made however to preserve wherever possible a distinction between rules that defined the law and those that applied it. The program listing is contained in Appendix B. This program has been implemented and run on the examples. At the end of the testing are the data and dialogues for each of the four examples. For completeness the examples are reproduced from the handbook in the Annex.

The previous stages of the study had revealed that the main problem lay in providing a powerful representation of events and time periods. The SSP legislation, as the first rule shows, is all about time periods and the relationships between them. What was needed was a set of algorithms to complement the definitions. Just as the 'sort' example at the start of this chapter is essentially useless for problem solving, so general definitions of periods and events had no power to solve problems. Analysis of the type of relationships between periods also showed that it would be necessary to have different procedures for finding and for 'showing' problems. Procedures that could show that a number of days satisfied some rules would be combinationally explosive, or non terminating if used to find a day. In most cases the procedures could only be used one way because the operations entailed arithmetic operations or tests that could only instantiate in one direction.

A uniform set of procedures for representing and manipulating time periods was therefore designed. Study of the legislation derived that a period could be defined by:

(1) specific days, eg week begins on Sunday;

(2) specific dates, eg tax year begins on 4 April;

(3) some condition true on every day, and not on adjacent days, eg sickness;

(4) restrictions on length (max or min) of a period defined by any of 1 to 3;

(5) derivation or intersection of periods defined by 1 to 4;

(6) linking of periods defined by 1 to 5, eg piw derived form sickness with gaps less than 15 days

(7) restrictions on length of a period defined by 1 to 6, eg a contract is linked and must have aggregate length exceeding 13 weeks;

(8) definitions 1 to 7 plus conditions that must be satisfied on the first day, eg period of entitlement. The conditions do not necessarily pertain to that day directly, eg the entitlement conditions refer to the contract length and other benefits received in the preceding 6 weeks;

(9) definitions 1 to 8 plus terminating events, eg imprisonment terminates entitlement;

(10) cumulative condition, eg sum of SSP received exceeding limit terminates entitlement.

The representation designed allows all these types of definition to be handled uniformly except for the last. The definition of when the entitlement limit is reached is interesting in that it is algorithmic. For a day of sickness a daily rate is calculated and paid in full. Because this daily rate depends on the number of qualifying days in the week it is not constant so the maximum amount a person can receive is up to the maximum for one day over the 'entitlement limit'. The day on

which the sum recieved exceeds the limit is the last day of entitlement. In order to express this, and the more complicated case where the weekly rate of SSP changes along the way, the legislation has to be given algorithmically. To have implemented an equivalent algorithm in this project would have been very time consuming and did not seem justified, so the maximum entitlement rules have not been completed.

A small number of predicates are used to define the characteristics of each type of period found in the legislation. All the period definitions may be found in the second section of the program listing. The definitions for a piw are as follows:

```
period_type(piw,linked).
period_sub_period_name(piw,sub_piw).
period_linkage(piw,14).

period_type(sub_piw,derived).
period_derivations(sub_piw,[sickness]).
period_min_length(sub_piw,4).
```

The sickness period is then defined:

```
period_type(sickness,primitive_condition).
period_definition(sickness,day_of_sickness).
```

To handle these standardised forms of definition a small number of procedures are defined, found in the third section of the listing. There are two basic searching operations that are needed both for constructing

a period from its definition and for answering the example queries. The first takes a day and the name of a period, and finds the period of the named variety that includes the day:

```
period_including_day(Period,Period_name,Day).
```

The data representation used for a period is a list of all the days in it. This allows linked periods to be handled as easily as non-linked ones. If no period meeting the conditions is found then Period is instantiated to the empty list. This procedure uses lower level ones that can generate dates forwards and backwards from the day of interest and uses the period definitions to test for inclusion. To determine a sickness period for example the day_of_sickness predicate is used until a day is found which fails. Having built up a period, limiting conditions are then applied as may be seen from the top level procedure in the listing. The second basic procedure for building up time periods takes a start and end date (a time frame) and searches for periods of a defined type within that time frame. Two versions are required, one to search from the start forwards, the other from the end backwards,

```
period_within_time_frame_forward(P, Period_name,T_start,T_end).
```

This is written in such a way that when backtracking occurs it will find the next solution, returning the empty list when there are no more. This procedure is able to use the first one together with a few rules to adjust its time frame whenever a solution is found. For the rules that handle periods as part of other definitions it is useful to add one further procedure that collects up all the solutions within a time frame:

all_periods-within_time_frame_forward(P,Period_name,T_start,T_end).

Looking now at the first two sections of the listing we can see a clear distinction has been maintained between those rules that define the law and those that apply the definitions. The latter must know about the data structures for periods and the procedures to handle them, whereas the former do not. In each section the application rules are identified by a subheading. For both of these types of rules the logic programming formalism appears to be natural and concise. It is only for the basic period searching procedures that the logic is less appropriate. The basic operations entail searching up and down lists in an efficient way that is more conveniently handled by an algorithmic language. Designing these procedures in logic was considerabley more difficult than writing the other rules, and getting them to work was definitely a process of debugging rather than 'knowledge engineering'. The efficiency implications cannot be ignored either: the system as written could not handle the fourth of the example cases in the form in which it was stated because it ran out of stack space in very deep recursions involving very large data structures. No doubt some ingenuity and a compiler that optimises tail recusion would help, but the conclusion that logic is not ideal for this type of operation appears inescapable.

At the end of the program there is a listing of the bottom level goals. It is apparent that these are all direct equivalents of conditions found in the legislation, rather than invented to fill problem solving gaps in it. It would be entirely reasonable to declare all these to be 'askable' of the user. By establishing a conceptual framework for the problem solving, albeit a simple one, we have put the problem solving back into the system.

As an example of the program in action we give below a slightly abbreviated trace of the solution of a simple query from example 1. The query is to determine the appropriate weekly rate of ssp on a particular day:

```
? weekly_rate_of_ssp(310,Weekly_rate).
Weekly_rate=37
```

The rate is as follows with depths of indentation indicating the depth of the subgoal search:

```
normal_weekly_earnings(310,E)
 normal_weekly_earnings_by_two_pay_days(310,E)
  two_preceding_pay_days_separated_by_eight_weeks(310,P1,P2)
   period_within_time_frame_backward([P2],pay_day,0,310)
   P3 is P2-1
   period_within_time_frame_backward_([P1],pay_day,0,P3)
   gap_between_periods(P1,P2,Gap)
   Gap>=55
  P3 is P1+1
  pay_received_in_period(P3,P2,Pay)
   all_periods_within_time_frame_forward(pay_days,day_of_payment,
                                                          P1,P2)
   sum_pay_received(Pay_days,Pay)
  earnings_calculated_from_two_pay_days(P1,P2,Pay,E)
```

This simple example shows the use of several different searches through dates to establish the relevant pay days and all the pay received in the period between them.

One simplification adopted in this program design should be noted at this point. There is an implicit 'point of view' when talking about time periods. If the current data is in the middle of, say, a period of sickness, then the end date of that period is strictly not known. Similarly, if a contract end date has not been specified but simply lies some time in the future then it is irrelevant for consideration of events up to the current date. A complete handling of time periods for our system ought to include this 'point of view' explicitly. This was not done because of the added complexity and because minor adjustments to the way the example data was presented obviated any problems.

The foregoing remarks allow us to see how we could set about achieving a more sensible dialogue structure for our problem solver. Take as an example the definition of a period of entitlement. This is the most complicated period definition but its interesting feature at this point is that the period is ended by the first to occur of a number of events: end of contract, start of pregnancy disqualifying period, start of a period of legal custody, or end of piw (the usual case). The problem solving rules first of all construct the intersection of the current contract and the piw; this should be done with reference to the current date as an end date as noted above. This establishes a period within which to search for the first instance of any of the other terminating events. At this point it would not be sensible for the system to ask of each day in turn whether it satisfies those conditions - it should discover whether the event occurred in the period of interest, ie it should generalise the question in the way that a human questioner would. In order for this to be possible it would only be necessary to introduce a new relation:

known(Period_name,Start,Finish)

that would record the period of time for which the data currently available to the system represented a complete history. Then an askable goal would only be asked if its history were not known for the period in question. If it were asked then the 'known' relation would require updating. Without a mechanism such as this we are not able to cope with the statement in example 2, where it says there is no reason to suspect the (27 year old female) employee is pregnant. PROLOG cannot represent negation directly, ie we cannot include 'not pregnant' as an assertion to be used by the theorem prover. Negation is implied from failure to prove a goal. By including the assertion:

known(pregnancy,_,_).

we would prevent further attempts to establish facts relating to pregnancy. Even if we had explicit negation, the 'known' predicate would serve to guide the process of generalising questions. An implementation of this idea was beyond the scope of this project, but we note that two approaches could be taken. One would use PROLOG intrinsics to assert and retract versions of the 'known' clause, essentially doing assignments; the other would build a declarative model of database update within which the whole system could operate.

Although we have gone a small way towards giving our system problem solving capabilities and some possibility of improving its dialogue at a rudimentary level, we have not yet tackled representing the definitional aspect of the law for consultation, distinct from problem solving. In

example 1 the system can correctly apply the rules that allow an employer to withold payment in lieu of a waiting day for which notification of sickness was not given, but it has no means to answer the question actually posed: "What can you do about the late notification?". As another example, consider the rules defining a 'day of sickness' that were discussed in the previous section. We saw there how a dialogue could be contrived to take a user through the detailed definition and then ask a generalising question. This was really a trick, combining the two types of consultation and clearly does not carry over into our new system. Once we have a conceptual framework for some part of the legislation however we can incorporate procedures to explain that framework. As a trivial example, if we redefined our period predicates to look like:

```
period(Period_name,type,x)
period(Period_name,derivation,y)
  etc
```

then a query of the form

```
? - period(sickness,X,Y).
```

would recover the definitional predicates. Clearly we would want to have some more sophisticated retrieval that could recursively unravel all the dependencies and explain the relationship between them, but the point is that by having a conceptual structure related to a user's concept we have the germ of a consultation system on the concept of the legislation. It is worth recapping here the observations made in section 2.7 above

concerning the need to establish the basic level of justification to which a concept must be reduced, and that this level depends on the class of users. The level of justification for periods is simple because it rests on common world concepts, it will not be so simple for, eg rights and obligations. Our system has no framework for making explicit the notions of what the employer and employee must do in order that the legislation is a true model of real world events. It therefore has nothing that by simple extension could answer the 'late notification' query, and is totally unable to deal in any useful way with the example 2.

As a final point on our conceptual framework we note that even when we are dealing with such comparatively simple things as time periods and events the representation is quite complex. The nature of the law is to always introduce exceptions to general rules; the system allows for this in its representation of periods by including 'special conditions' that can be applied after everything else. Even this however does not deal with the rule that says that the end of a contract does not terminate a period of entitlement if the contract was terminated solely or mainly to avoid liability for SSP. To have accommodated this would require the ability to attach whole clauses where we only had predicate names. To introduce it as an afterthought would require substantial reworking. We note that this would be a traditional 'programming' rather than 'knowledge engineering' task. The modifiability of rules pertains only to the direct expression of the legislation, and we have now established just how much and how little that can do for us.

4.5 SUMMARY

In summary therefore we observe:

(1) A useful consultation system on any large scale will have to make explicit knowledge of four types: definitional, problem solving, dialogue, and written structure. This knowledge is not made explicit by the direct representation of the legislation in logic.

(2) A conceptual structure is a sine qua non for consultation on definitions and anything beyond trivial problem solving.

(3) Problem solving may be separated into the application of low level procedures to general definitions. The interfacing procedures are naturally written in logic, but the low level ones less so.

* * * *

CHAPTER V

CONCLUSIONS

This project started with the perception of a match between a technique, logic for rule based systems, and the domain of legislation, and sought to investigate how far the technique would go in helping to build a knowledge based system of power and flexibility. The method was practical, the representation of a new piece of legislation as a means of gaining insight into the problems involved. The conclusions arise from making a number of distinctions that are substantiated by a survey of the literature and which help us to assign the correct role for logic in the building of knowledge based systems.

The most important distinction is the one that we took as a framework for the survey of knowledge representation: that an intelligent system must be described at two levels, the knowledge level and the symbol level. This enabled our study of logic for problem solving to make clear that we must not confuse power at the representational level with competence at the knowledge level. The detailed analysis of MYCIN by Clancey further demonstrated how only by analysing each competence we require of the system can we make the necessary knowledge explicit; until we have made it explicit we cannot represent it; and until we have represented it we have not given that competence to our system, whatever the representational formalism. Clancey's paper also demonstrated that our epistemological studies might have implications for improving the domain

independent deductive strategies we implement for our logic systems. We would only gain benefit from such improvements in those domains where we could provide the corresponding structural knowledge for the deductive rules to act upon.

An interesting question left unresolved by the survey of knowledge representation was the status of semantic nets and their interpreters as specialised inference tools tailored to specific uses. It remains an open question whether for an equivalent logic representation we can always use the separation of an algorithm into logic and control to optimise the control component adequately by automatic means. It seems likely that the current state of the art is that it is a theoretical possibility rather than one available to programmers. However, a detailed study would appear to be a useful impetus to work in program transformation.

Another area that it becomes apparent could be fruitfully studied from a logic programming viewpoint was the use of frames seen as hypothesis generation mechanisms and a notation for handling collections of clauses. The frame notion when separated from its connection with semantic networks appears to be more an expression of epistemological insight at the knowledge level rather than an essential representational tool.

Our general conclusions on logic as a representational formalism are therefore that its power has still not been fully explored in a number of important areas, and that we should not let those explorations distract us from our other task, the study of the knowledge level itself.

Turning to the experience of building a system we found the lessons of Clancey's study were just as relevant despite the seeming ease of translating legislative rules into logic rules. We found that in order to build a system with any competence we had to be specific about the competence required - was it to explain definitions, reason about cases, hold sensible dialogues with a user, etc? With a clear purpose in mind the logic programming method proved to be powerful and easy to use in all respects except the low level manipulation of data structures. we found that even a simple problem solving system involved a knowledge acquisition task, ie deriving some underlying real world concepts pertaining to the tasks. We also noted that the easy modifiability of the rules that represent the legislation would not carry over quite so easily into our more powerful conceptual model. In other words there is a good deal of programming as well as knowledge engineering in building a knowledge based system in a complex domain. It was apparent that extending a system to deal with all the law, instead of just the parts displaying a high degree of structure over a few basic concepts, would require conceptual models of far more detail than we have at present. The TAXMAN example showed just how far one would expect to go away from the original expression of the law in order to model the concepts adequately. The written form of the law, as a collection of cross referencing statements, was also uncovered as requiring separate explicit treatment in the future.

Our general conclusion from this study is therefore that the match between the technique and the domain exists only at a surface level and is perhaps inclined to deceive us into believing that in this domain the epistemology of the knowledge level comes free with the knowledge - we have demonstrated that that is not the case.

REFERENCES

1. A programme for Advanced Information Technology. The Report of the Alvey Committee. HMSO 1982.

2. Addis, T.R. (1982). Expert Systems. An Evolution in Information Retrieval. Information Technology. Research and Development (1982) Vol 1, No.4, pp 301-324.

3. Addis T.R. & Johnson, L. (1982). Knowledge for Machines. Brunel University MCSG/TR25.

4. Allen, L (1979). Language, Law and Logic. Plain Legal Drafting for the Electronic Age. In Niblett, ibid.

5. Bellord, N. (1979). Tax Planning By Computer. In Niblett, ibid.

6. Brachman, R.J. (1979). On the Epistemological Status of Semantic Networks in Findler, ibid.

7. Brachman, R.J., R. Smith, B.C. (1980). Special issue on Knowledge Reproduction SIGART 70, pp 1-138.

8. Ciampi, C. (Ed), (1982). Artificial Intelligence and Legal Information Systems. North Holland.

9. Clancy, W.J. (1983). The Epistemology of a Rule-Based Expert System. Artifical Intelligence, Vol 20, No 3. pp 215-251.

10. Clark, K and McCabe, F.G. (1982). In Hayes, J & Michie D.J. (Eds) Machine Intelligence, Ellis and Horwood.

11. du Feu, D. (1980). Selecting Welfare Benefits by Computer. In Niblett, ibid.

12. Employer's Guide to Statutory Sick Pay. (1982). DHSS. NI.227.

13. Findler, N.V. (Ed), (1979). Associative Networks, Representation and Use of Knowledge by Computers. Academic Press.

14. Hammond, P. (1983). Representation of DHSS Regulations as a Logic Program. In proceedings of Expert Systems 83, pp 225 -235.

15. Hammond, P. APES: A user manual. Imperial College Research Report DOC 82/9. Imperial College, London.

16. Hawkinson, L (1975). The Representation of Concepts in OWL. Proceedings of the 4th International Joint Conference on Artificial Intelligence, 1977 pp 67-76.

17. Hellawell, R (1980). A Computer Program for Legal Planning and Analysis. Taxation of Stock Redemptions. Columbia Law Review 80 (7), pp 1363-1398.

18. Hendrix, G.G. (1979). Encoding Knowledge in Partitioned Networks in Findler, ibid.

19. Hewitt, C. (1969). PLANNER. A Language for Proving Theorems in Robots. Proc. IJCAI-I pp 295-301.

20. Intelligent Knowledge Based Systems. A Programme for Action in the U.K. (1983). SERC.

21. Kowalski, R. (1979). Logic for Problem Solving. North Holland.

22. Kowalski, R. (1983). Logic for Expert Systems. In Procedings of Expert Systems 83, pp 79-93. BCS.

23. McCarty, T. (1980). The TAXMAN Project: Towards a Cognitive Theory of Legal Argument. In Niblett, ibid.

24. McCarty, T (1982). Intelligent Legal Information Systems: Problems and Prospects. In Proc. Coll. on Data Processing and the law. UK Committee of Comparative law.

25. McCarty, T. Permissions and Obligations. In Proc IJCAI 83 pp 287-294.

26. Newell, A. (1982). The Knowledge Level. Artificial Intelligence, Vol 18, No. 1, pp 87-127.

27. Niblett, B. (Ed), (1980). Computer Science and Law. Cambridge University Press.

28. Philips, B (1978). A Model for Knowledge and its Application to Discourse Analysis. Am. J. Computational Linguistics. Vol 5 (4).

29. Robinson, J.A., (1965). A Machine Oriented Logic Based on the Resolution Principle. JACM 12 pp 23-41.

30. Schank, R.C. (1975). Conceptual Information Processing. North Holland.

31. Schank, R.C. (1979). Re: The Gettysburg Address. In Findler, ibid.

32. Sergot, M. (1980). Legol as a Logic Programming Language. Imperial College, London.

33. Sergot, M. (1983). A Query-the-User facility for logic programming. In Degano, P. & Sandwell, E (Eds), Intergrated Interactive Computer Systems. North Holland.

34. Shortliffe, E.H. (1976). Computer-Based Medical Consultations: MYCIN. Elsevier, New York.

35. Social Security and Housing Benefits Act 1982. HMSO.

36. Stamper, R.K. (1973). The LEGOL Project and Languages. Proc Datafair Conference. BCS.

37. Stamper, R.K. (1979). Towards a Semantic Normal Form. IFIP TC2 Working Conference on Database Architecture, Venice.

38. Stamper, R. (1980). LEGOL: Modelling Legal Rules by Computer. In Niblett, ibid.

39. Statutory Sick Pay (General) Regulations 1982. HMSO.

40. Sussman, G.J. & McDermott, D.V. (1972). Why Conniving is Better than Planning. AI Memo No 255, MIT Project Mac.

41. Tannenbaum, A.S. (1976). Structured Computer Organisation. Prentice Hall.

42. Winograd, T. (1975). Frame Representations and the declarative/procedural controversy. In Bobrow D.G. and Collins A, Eds, Representation and Understanding pp 185-210. Academic Press.

APPENDIX A


```
% ************************************************************
%
%       SSP_ASK
%
%       SSP consultation program without time period
%       modelling.
%
%       W P Sharpe
%       May 1984
%
% ************************************************************

entitled_to_ssp_for_day :-
        day_is_part_of_a_piw,
        day_is_within_a_period_of_entitlement,
        day_is_a_qualifying_day,
        not(day_is_excluded).

day_is_part_of_a_piw :-
        day_is_a_day_of_sickness,
        day_is_one_of_four_or_more_days_sickness.


day_is_a_day_of_sickness :-
        under_medical_care_in_respect_of_disease_or_disablement,
        has_done_no_work_under_the_contract_of_service_on_day.



day_is_a_day_of_sickness :-
        stated_by_a_registered_medical_practitioner_that_he_should_not_work_for_day,
        has_done_no_work_under_the_contract_of_service_on_day.


day_is_a_day_of_sickness :-
        at_commencement_of_day_or_during_the_day_he_became_incapable_of_work_specific_to_contract_of_service,
```

```
        has_done_no_work_under_the_contract_of_service_on_day_except_during_a_shift_which_ends_on_that_day_hav



day_is_within_a_period_of_entitlement :-
        relevant_date(R),
        not(terminated_by_maximum_entitlement(R)),
        not(terminated_by_contract(R)),
        not(terminated_by_legal_custody(R)),
        not(terminated_by_pregnancy(R)),
        not(entitlement_disqualified_by_schedule_one_for_relevant_date(R)).

relevant_date(R) :-
        beginning_of_piw(S),
        beginning_of_most_recent_contract(C),
        later(S,C,R).

beginning_of_piw(P) :-
        period_of_sickness_including_day_in_question_began(S),
        period_of_linked_sickness_began(S, P).


period_of_linked_sickness_began(S, P) :-
        start_of_period_sickness_greater_than_four_days_ending_within_previous_fourteen_days(S, P1),
        period_of_linked_sickness_began(S1, P).

period_of_linked_sickness_began(S, S).



later(A, B, C) :-
        B>A, !.
later(A, B, A).

terminated_by_maximum_entitlement :-
```

```
        entitlement_limit(Limit),
        entitlement_exceeds_limit_for_period_or_tax_year(Limit).

entitlement_limit(Limit) :- ???

entitlement_exceeds_limit_fore_period_or_tax_year(Limit) :-
        total_entitlement_for_current_period_of_entitlement_up_to_and_including_day_in_question(E),
        E >= limit.

total_entitlement_for_current_period_of_entitlement_up_to_and_including_day_in_question(E) :- ???

entitlement_exceeds_limit_fore_period_or_tax_year(Limit) :-
        total_entitlement_for_current_tax_year_up_to_and_including_day_in_question(E),
        E >= Limit.

total_entitlement_for_current_tax_year_up_to_and_including_day_in_question(E) :- ???

terminated_by_contract(F) :-
        most_recent_contract_ended_before_day_in_question,
        not(contract_was_terminated_solely_or_mainly_to_avoid_liability_for_ssp,
        contiact_due_to_expire_on_or_after_day_in_question).



terminated_by_legal_custody(R) :-
        detained_in_legal_custody_between_relevant_date_and_day_in_question(R).


terminated_by_legal_custody(R) :-
        sentenced_to_imprisonment_between_relevant_date_and_day_in_question(R),
        not(sentence_is_suspended).


terminated_by_pregnancy(R) :-
        pregnant_in_preceding_seven_months(R),
```

```
        expected_week_of_confinement_commencement(C)
        D is C-78,
        D >= R,
        precedes_day_in_question(D).



entitlement_disqualified_by_schedule_one_for_relevant_date(R) :-
        employee_is_over_pensionable_age_on_relevant_date(R);
        disqualified_by_short_contract_on_relevant_date(R);
        normal_weekly_earnings_below_lower_limit(R);
        disqualified_by_pension_day_in_interruption_of_employement_preceding_relevant_date(R);
        disqualified_by_sickness_benefit_entitlement_in_preceding_period(R);
        disqualified_by_maternity_allowance_in_preceding_period(R);
        disqualified_by_reason_of_having_done_no_work_under_contract(R);
        disqualified_by_trade_dispute(R);
        reached_maximum_tax_year_entitlement_before_relevant_date(R);
        employee_is_or_has_been_pregnant_with_a_disqualifying_period_including_relevant_date(R);
        disqualified_by_legal_custody(h).


disqualifed_by_short_contract_on_relevant_date(R) :-
        contract_for_day_in_question_was_entered_into_for_a_specified_period_of_not_more_than_three_months,
        not(special_short_contract_conditions_apply(R)).


special_short_contract_conditions_apply(R) :-
        contract_of_service_has_become_a_contract_for_a_period_exceeding_three_months_on_relevant_date(R), !.

special_short_contract_conditions_apply(R) :-
        current_contract_was_preceded_by_a_contract_with_same_employer_which_ceased_to_have_effect_not_more_th
urrent_contract_started,
        aggregate_length_of_preceding_contracts_with_current_contract_exceeds_thirteen_weeks.


aggregate_length_of_preceding_contracts_with_current_contract_exceeds_thirteen_weeks :-
```

```
        specified_period_of_current_contract_on_day_in_question(C),
        period_of_earlier_contract(P),
        A is C+P,
        accumulated_aggregates_exceed_thirteen(A).



accumulated_aggregates_exceed_thirteen(A) :-
        A > 13,
        !.

accumulated_aggregates_exceed_thirteen(A) :-
        earlier_contract_terminated_within_eight_weeks_of_those_already_considered,
        period_of_earlier_contract(L),
        Agg is A + L,
        accumlated_aggregates_exceed_thirteen(Agg).


normal_weekly_earnings_below_lower_limit(R) :- ???

disqualified_by_pension_day_in_interruption_of_employment_preceding_relevant_date(R) :-
        one_day_in_57_preceding_relevant_date_formed_part_of_a_period_of_interruption_of_employment(R),
        invalidity_pension_day_occured_during_period_of_interruption_of_employment.


invalidity_pension_day_occured_during_period_of_interruption_of_employment :-
        day_during_period_of_interruption_of_employment_for_which_employee_was_entitled_to_an_invalidity_or_no


invalidity_pension_day_occured_during_period_of_interruption_of_employment :-
        day_during_period_of_interruption_of_employment_for_which_employee_was_not_entitled_to_an_invalidity_o
ion_but_which_was_the_last_day_of_invalidity_pension_qualifying_period.


disqualified_by_sickness_benefit_entitlement_in_preceding_period(R) :-
        one_day_in_57_preceding_relevant_date_was_day_of_entitlement_to_sickness_benefit_or_would_have_been_if
s_satisfied(R).
```

```
disqualaifed_by_maternity_allowance_in_preceding_period(R) :-
        one_day_in_57_preceding_relevant_date_was_day_of_entitlement_to_maternity_allowance(R).


disqualified_by_reason_of_having_done_nc_work_under_contract(R) :-
        on_relevant_date_employee_had_done_no_work_under_contract_of_service(R),
        not(employee_had_worked_on_earlier_linked_contract(R)).


employee_had_worked_on_earlier_linked_contract(R) :-
        an_earlier_contract_ceased_to_have_effect_not_more_than_eight_weeks_before_the_contract_valid_on_relev
        employee_has_worked_on_earlier_contract.


disqualified_by_trade_dispute(R) :-
        on_relevant_date_there_was_a_stoppage_of_work_due_to_a_trade_dispute_at_the_employees_place_of_employm
        not(employee_proves_at_no_time_on_or_before_relevant_date_did_he_have_direct_involvement_or_interest_i
tion(R)).


reached_maximum_tax_year_entitlement_before_relevant_date(R) :- ???.

employee_is_or_has_been_pregnant_with_a_disqualifying_period_including_relevant_date(R) :-
        pregnant_in_preceding_seven_months(R),
        expected_week_of_confinement(C),
        D is C-77,
        R >= D,
        not(pregnancy_terminated_before_disqualifying_period_otherwise_than_by_confinement(D)).


disqualfied_by_legal_custody(R) :-
        detained_in_legal_custody_at_any_time_on_relevant_date(R).

disqualfied_by_legal_custody(R) :-
```

```
        sentenced_to_imprisonment_on_relevant_date(R),
        not(sentence_is_suspended).


day_is_a_qualifying_day :-

day_is_qualifying_day :-
        qualifying_days_are_agreed_days_of_week,!,
        day_is_agreed_or_obligatory_agreed_qualifying_day.


days_is_agreed_or_obligatory_agreed_qualifying_day :-
        week_beginning_sunday_and_containing_day_in_question_includes_at_least_one_agreed_qualifying_day,
        !,
        day_in_question_is_an_agreed_qualifying_day.



day_is_agreed_or_obligatory_qualifying_day :-
        day_is_obligatory_qualifying_day_for_week.


day_is_a_qualifying_day :-
        days_of_work_required_of_employee_are_agreed,
        !,
        day_is_agreed_or_default_day_of_work.


day_is_agreed_or_default_day_of_work :-
        week_beginning_sunday_and_including_day_in_question_includes_at_least_one_day_of_work,
        !,
        day_in_question_is_an_agreed_day_of_required_work.

day_is_an_agreed_or_default_day_of_work :-
        day_in_question_is_a_wednesday.
```

```
day_is_a_qualifying_day :-
        not(day_in_question_is_a_day_on_which_no_employees_expected_to_work).



day_is_excluded :-
        day_is_in_first_three_qualifying_days;
        day_is_excluded_by_entitlement_limit;
        day_is_excluded_for_failure_to_notify_sickness.


day_is_excluded_by_entitlement_limit :- ???.

day_is_ecluded_for_failure_to_notify_sickness :-
        day_may_be_excluded_for_failure_to_notify_sickness,
        employer_has_exercised_right_to_withhold_payments_in_respect_of_unnotified_day.

day_may_be_excluded_for_failure_to_notify_sickness :-
        not(notification_of_sickness_given_for_day_in_question).


day_may_be_excluded_for_failure_to_notify_sickness :-
        notification_of_sickness_given_for_day_in_question,
        number_of_waiting_days_for_which_notification_was_not_given(W),
        number_of_qualifying_days_between_waiting_days_and_day_in_question_for_which_notification_given(N),

        M is N+1,
        N == M.


day_is_excluded_by_entitlement_limit :-
        entitlement_limit(L), ???
```

```
% ************************************************************
%
%       Goals that must be defined externally for SSP_ASK
%
% ************************************************************

askable(day_is_one_of_four_or_more_days_sickness).
askable(under_medical_care_in_respect_of_disease_or_disablement).
askable(has_done_no_work_under_the_contract_of_service_on_day).
askable(stated_by_a_registered_medical_practitioner_that_he_should_not_work_for_day).
askable(at_commencement_of_day_or_during_the_day_he_became_incapable_of_work_specific_to_contract_of_service).
askable(has_done_no_work_under_the_contract_of_service_on_day_except_during_a_shift_which_ends_on_that_day_hav
.
askable(period_of_sickenss_including_day_in_question_began).
askable(start_of_period_sickness_greater_than_four_days_ending_within_previous_fourteen_days).
askable(beginning_of_most_recent_contract).
askable(most_recent_contract_ended_before_day_in_question).
askable(contrace_was_terminated_soleley_or_mainly_to_avoid_liatility_for_ssp).
askable(contract_due_to_expire_on_or[_after_day_in_question).
askable(detained_in_legal_custody_between_relevant_date_and_day_in_question).
askable(sentenced_to_imprisonment_between_relevant_date_and_day_in_question).
askable(sentence_is_suspended).
askable(pregnant_in_preceding_seven_months).
askable(expected_week_of_confinement_commencement).
askable(proceedes_day_in_question).
askable(employee_is_over_pensionable_age_on_relevant_date).
askable(contract_for_day_in_question_was_entered_into_for_a_specified_period_of_not_more_than_three_months).
askable(current_contract_was_preceded_by_a_contract_with_same_employer_which_ceased_to_have_effect_not_more_th
urrent_contract_started).
askable(specified_period_of_current_contract_on_day_in_question).
askable(period_of_earlier_contract).
askable(one_day_in_57_preceding_relevant_date_formed_part_of_a_period_of_interruption_of_employment).
askable(day_during_period_of_interruption_of_employment_for_which_employee_was_entitled_to_an_invalidity_or_no
.
askable(day_during_period_of_interrutpion_of_employment_for_which_employee_was_not_entitled_to_an_invalidity_o
ion_but_which_was_the_last_day_of_invalidity_pension_qualifying_period).
askable(one_day_in_57_preceding_relevant_date_was_day_of_entitlement_to_sickness_benefit_or_would_have_been_it
s_satisfied).
askable(one_day_in_57_preceding_relevant_date_was_day_of_entitlement_to_maternity_allowance).
askable(on_relevant_date_employee_had_done_no_work_under_contract_of_service).
askable(an_earlier_contract_ceased_to_have_effect_not_more_than_eight_weeks_before_the_contract_valid_on_relev
```

```
askable(employee_has_worked_on_earlier_contract).
askable(on_relevant_date_there_was_a_stoppage_of_work_due_to_a_trade_dispute_at_the_employees_place_of_employm
askable(employee_proves_at_no_time_on_or_before_relevant_date_did_he_have_direct_involvement_or_interest_in_t
).
askable(pregnancy_terminated_before_disqualifying_period_otherwise_than_by_confinement).
askable(qualifying_days_of_week_are_agreed_days_of_week).
askable(week_beginning_sunday_and_containing_day_in_question_includes_at_least_one_agreed_qualifying_day)
askable(day_in_question_is_an_agreed_qualifying_day).
askable(day_is_obligatory_qualifying_day_for_week).
askable(days_of_work_required_of_employee_are_agreed).
askable(day_in_question_is_a_wednesday).
askable(day_in_question_is_a_day_on_which_no_employees_expected_to_work).
askable(day_is_in_first_three_qualifying_days).
askable(notification_of_sickness_given_for_day_in_question).
askable(number_of_waiting_days_for_which_notification_was_not_given).
askable(number_of_qualifying_days_between_waiting_days_and_day_in_question_for_which_notification_given).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                             %
%       SSP                                                   %
%                                                             %
%       A Prolog program to model the Statutory Sick Pay      %
%       legislation                                           %
%                                                             %
%       W P Sharpe                                            %
%       May 1984                                              %
%                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% ************************************************************
%
%       The following section contains general rules
%       not restricted to handling time periods.
%
% ************************************************************
%
%       ENTITLEMENT TO SSP
%

employee_is_entitled_to_ssp(Day) :-
        within_SSP_period(SSP_period, Day),
        not(excluded_from_ssp_period(SSP_period, Day)).

excluded_from_ssp_period(SSP_period, Day) :-
        waiting_day(SSP_period, Day).

excluded_from_ssp_period(SSP_period, Day) :-
        may_be_excluded_for_failure_to_notify(SSP_period, Day),
        employer_exercises_right_to_withhold_payment_for_failure_to_notify(Day).

may_be_excluded_for_failure_to_notify(_,Day) :-
        not(notification_of_sickness_given(Day)).
```

```
may_be_excluded_for_failure_to_notify(SSP_period, Day) :-
        may_be_excluded_for_waiting_day(SSP_period, Day, W).

may_be_excluded_for_waiting_day(SSP_period, Day, W) :-
        not(waiting_day(SSP_period, Day)),
        waiting_day(SSP_period, W),
        not(notification_of_sickness_given(W)),
        notification_of_sickness_given(Day),
        not(first_notification_sub_conditions(Day, W, SSP_period)),
        not(second_notification_sub_conditions(Day, W, SSP_period)).

first_notification_sub_conditions(Day, W, SSP_period) :-
        earlier_day_in_period(W1, W, SSP_period),
        may_be_excluded_for_waiting_day(SSP_period, Day, W1).

second_notification_sub_conditions(Day, W, SSP_period) :-
        earlier_day_in_period(Day1, Day, SSP_period),
        may_be_excluded_for_waiting_day(SSP_period, Day1, W).

waiting_day(SSP_period, W) :-
        in_first_three_days_of_period(SSP_period, W).

        %
        %       Application of rules to case
        %
within_SSP_period([S|T], Day) :-
        period_including_day([S|T], ssp_period, Day).

earlier_day_in_period(Day1, Day, P) :-
        time_frame(P, Start, End),
        E is Day-1,
        day_in_period_backward(Day1, Start, E).

in_first_three_days_of_period(Period, Day) :-
        append([_, _, Day], _, Period),
```

```
        in_period([D1, D2, D3], Day).

        %
        %       Problem solving rule to determine SSP entitlement
        %       within a given period
        %

ssp_due_in_period(Start, Finish, SSP_period, SSP) :-
        period_within_time_frame_forward(SSP_period, ssp_period,
        Start, Finish),
        ssp_sum_for_period(SSP_period, SSP).

ssp_sum_for_period([S|T], SSP) :-
        weekly_rate_of_ssp(S, Weekly_rate),
        ssp_sum([S|T], Weekly_rate, SSP).

ssp_sum([], _, 0).

ssp_sum([Day|T], Weekly_rate, SSP) :-
        daily_rate_of_ssp(Day, Weekly_rate, S1),
        ssp_sum(T, Weekly_rate, S2),
        SSP is S1+S2.

%
%       NORMAL WEEKLY EARNINGS
%

normal_weekly_earnings(Day, E) :-
        normal_weekly_earnings_by_two_pay_days(Day, E);
        normal_weekly_earnings_by_one_pay_day(Day, E);
        normal_weekly_earnings_by_no_pay_day(Day, E).

normal_weekly_earnings_by_two_pay_days(Day, E) :-
        two_immediately_preceding_pay_days_separated_by_at_least_eight_weeks(Day, P1, P2),
        P3 is P1+1,
        pay_received_in_period(P3, P2, Pay),
```

```
        earnings_calculated_from_two_pay_days(P1, P2, Pay, E).

earnings_calculated_from_two_pay_days(P1, P2, Pay, E) :-
        pay_day_intervals_are_multiples_of_calendar_months,
        nearest_whole_number_of_months(P1, P2, M),
        E is (Pay*12)/(M*52).

earnings_calculated_from_two_pay_days(P1, P2, Pay, E) :-
        not(pay_day_intervals_are_multiples_of_calendar_months),
        exact_number_of_weeks(P1, P2, W),
        E is Pay/W.

earnings_calculated_from_two_pay_days(P1, P2, Pay, E) :-
        not(pay_day_intervals_are_multiples_of_calendar_months),
        not(exact_number_of_weeks(P1, P2, W)),
        E is (Pay*7)/(P2-P1+1).

normal_weekly_earnings_by_one_pay_day(Day, E) :-
        not(two_immediately_preceding_pay_days_separated_by_at_least_eight_weeks(Day, P1, P2)),
        immediately_preceding_pay_day(Day, P),
        period_for_which_payment_received(P, Start, End),
        pay_received_in_period(Start, End, Pay),
        earnings_calculated_from_two_pay_days(Start, End, Pay, E).

normal_weekly_earnings_by_no_pay_day(Day, E) :-
        not(immediately_preceding_pay_day(Day, P)),
        normal_weekly_earnings_calculated_from_contract(E).

normal_weekly_earnings_calculated_from_contract(E) :-
        contractual_remuneration_is_weekly_entitlement(E).

normal_weekly_earnings_calculated_from_contract(E) :-
        contractual_remuneration_is_multiple_month_entitlement(M, Salary),
        E is (Salary*12)/(M*52).
```

```
        %       Normal weekly earnings - application to case
        %
two_immediately_preceding_pay_days_separated_by_at_least_eight_weeks(Day, P1, P2) :-
        period_within_time_frame_backward([P2], pay_day, 0, Day),
        !,
        P3 is P2-1,
        period_within_time_frame_backward([P1], pay_day, 0, P3),
        gap_between_periods([P1], [P2], Gap),
        Gap >= 55,
        !.

pay_received_in_period(P1, P2, Pay) :-
        all_periods_within_time_frame_forward(Pay_days, day_of_payment, P1, P2),
        sum_pay_received(Pay_days, Pay).

sum_pay_received([ ], 0).

sum_pay_received([Day|T], Pay) :-
        pay_history(Day, S1, _, _),
        sum_pay_received(T, S2),
        Pay is S1+S2.

nearest_whole_number_of_months(P1, P2, M) :-
        gap_between_periods([P1], [P2], G),
        M is (G+15)/30.

exact_number_of_weeks(P1, P2, W) :-
        gap_between_periods([P1], [P2], G),
        6 is G mod 7,
        W is (G+1)/7.

immediately_preceding_pay_day(Day, P) :-
        period_within_time_frame_backward([P], pay_day, 0, Day).

period_for_which_payment_received(Pay_day, Start, End) :-
        pay_history(Pay_day, _, Start, End).
```

```
%
%       WEEKLY RATE OF SSP
%

weekly_rate_of_ssp(Day, W) :-
        normal_weekly_earnings(Day, E),
        weekly_rate_for_earnings(E, W).

weekly_rate_for_earnings(E, 25) :-
        F is floor(E+1),
        F < 45.

weekly_rate_for_earnings(E, 31) :-
        F is floor(E+1),
        F >= 45,
        F < 60.

weekly_rate_for_earnings(E, 37) :-
        F is floor(E+1),
        F >= 60.

%
%       DAILY RATE FOR SSP
%

daily_rate_of_ssp(Day, S) :-
        weekly_rate_of_ssp(Day, W),
        number_of_qualifying_days_in_week(Day, Q),
        S is W/Q.

daily_rate_of_ssp(Day, Weekly_rate, S) :-
        number_of_qualifying_days_in_week(Day, Q),
        S is Weekly_rate/Q.

        %
```

```
        %       Application
        %

number_of_qualifying_days_in_week(Day, Q) :-
        period_including_day(Week, week, Day),
        time_frame(Week, Sun, Sat),
        all_periods_within_time_frame_forward(Q_days, qualifying_day, Sun, Sat),
        number_of_days_in_period(Q_days, Q).
```

```
% ************************************************************
%
%       The following section consists entirely of rules that
%       are used to define periods that may then be handled
%       in a uniform way.
%
% ************************************************************

%
%       PERIOD FOR WHICH SSP IS DUE
%

period_type(ssp_period, derived).
period_derivations(ssp_period, [piw, entitlement]).
period_special_condition(ssp_period, ssp_cond).

ssp_cond([ ], [ ]).

ssp_cond(P_cond, P) :-
        qualifying_days_in_period(P_qual, P),
        ssp_excluded_days(P_cond, P_qual).

qualifying_days_in_period(P_qual, P) :-
        time_frame(P, Start, End),
        all_periods_within_time_frame_forward(Q, qualifying_day, Start, End),
        intersection(P, Q, P_qual).

ssp_excluded_days(P_excl, SSP_period) :-
        sxd(P_excl, [ ], SSP_period, SSE_period).

sxd(P, P, [ ], _).

sxd(P_excl, Part, [First|Rest], SSP_period) :-
        excluded_from_ssp_period(SSP_period, First),
        !,
        sxd(P_excl, Part, Rest, SSP_period).
```

```
sxd(P_excl, Part, [First|Rest], SSP_period) :-
        append(Part, [First], P),
        sxd(P_excl, P, Rest, SSP_period).


%
%       WEEK
%

period_type(week, framing).
period_definition(week, sunday).
period_length(week, 7).

%
%       TAX YEAR
%

period_type(tax_year, framing).
period_definition(tax_year, april_fourth).
period_length(tax_year, 365).

        % Assumed reference definition for dates is that the first day
        % of a calender year is 1 and all problem dates are later.

april_fourth(Day) :-
        94 is Day mod 365.

%
%       SICKNESS
%

period_type(sickness, primitive_condition).
period_definition(sickness, day_of_sickness).

%
```

```
%       PERIOD OF INCAPACITY FOR WORK (PIW)
%

period_type(piw, linked).
period_sub_period_name(piw, sub_piw).
period_linkage(piw, 14).

period_type(sub_piw, derived).
period_derivations(sub_piw, [sickness]).
period_min_length(sub_piw, 4).

%
%       PERIOD OF ENTITLEMENT
%

period_type(entitlement, derived).
period_derivations(entitlement, [piw, entitling_contract]).
period_exclusions(entitlement, [preg_disqual, legal_custody]).
period_first_day_conditions(entitlement,
        [below_pensionable_age,
%       contract_over_minimum_length,
        normal_weekly_earnings_above_lower_limit,
        no_pension_day_in_interruption_of_employment_in_preceding_57_days,
        no_sickness_benefit_entitlement_in_preceding_57_days,
        no_maternity_allowance_entitlement_in_preceding_57_days,
%       work_done_under_contract,
%       not_disqualified_by_trade_dispute,
%       below_maximum_tax_year_entitlement,
        not_within_pregnancy_disqualifying_period]).


below_pensionable_age(D) :-
        age(Date, Age),
        A is Age + (D-Date)/365,
        A < 65.
```

```
normal_weekly_earnings_above_lower_limit(Day) :-
        normal_weekly_earnings(Day, E),
        E > 29.

no_pension_day_in_interruption_of_emplcyment_in_preceding_57_days(Day) :-
        Day1 is Day - 57,
        period_within_time_frame_backward([ ], state_benefit, Day1, Day).

no_sickness_benefit_entitlement_in_preceding_57_days(Day) :-
        Day1 is Day - 57,
        period_within_time_frame_backward([ ], state_benefit, Day1, Day).

no_maternity_allowance_entitlement_in_preceding_57_days(Day) :-
        Day1 is Day - 57,
        period_within_time_frame_backward([ ], state_benefit, Day1, Day).


not_within_pregnancy_disqualifying_period(Day) :-
        period_including_day([ ], preg_disqual, Day).

%
%       ENTITLING CONTRACT
%
period_type(entitling_contract, linked).
period_sub_period_name(entitling_contract, sub_entitling_contract).
period_linkage(entitling_contract, 56).
period_min_length(entitling_contract, 91).

period_type(sub_entitling_contract, primitive_event).
period_definition(sub_entitling_contract, contract_period).

%
%       PREGNANCY DISQUALIFYING PERIOD
%

period_type(preg_disqual, derived).
```

```
period_derivations(preg_disqual, [expected_confinement_as_defined_by_Act]).
period_special_condition(preg_disqual, pregnancy_disqualification).

period_type(expected_confinement_as_defined_by_Act, event).
period_definition(expected_confinement_as_defined_by_Act, expected_confinement).

pregnancy_disqualification([ ], [ ]) :- !.

pregnancy_disqualification(P_disqual, [C|_]) :-
        Start is C - 77,
        End is C + 48,
        construct_period_from_limits(P_disqual, Start, End).


%
%       STATE BENEFIT
%
%       For the purposes of this pilot project the various benefits
%       (pension, maternity, sickness) that can remove entitlement
%       to SSP under Schedule 1 and lumped together under the
%       definition "state benefit"
%

period_type(state_benefit, primitive_event).
period_definition(state_benefit, state_benefit).

%
%       LEGAL CUSTODY
%

period_type(legal_custody, primitive_event).
period_definition(legal_custody, legal_custody).

%
%       QUALIFYING DAYS
%
```

```
period_type(qualifying_day, event).
period_definition(qualifying_day, qualifying_day).

qualifying_day(D) :-
        qualifying_days_are_agreed,
        agreed_qualifying_day(D).

qualifying_day(D) :-
        qualifying_days_are_determined_by_regulations,
        qualifying_day_by_regulations(D).

qualifying_days_are_determined_by_regulations :-
        not(qualifying_days_are_agreed).

qualifying_day_by_regulations(D) :-
        contractual_working_days_are_agreed,
        contractual_working_day(D).

qualifying_day_by_regulations(D) :-
        contractual_working_days_are_agreed,
        week_including_day_contains_no_agreed_contractual_working_days(D),
        obligatory_qualifying_day_of_week(D).

qualifying_day_by_regulations(D) :-
        not(contractual_working_days_are_agreed),
        actual_working_days_are_agreed,
        actual_working_day(D).

qualifying_day_by_regulations(D) :-
        not(contractual_working_days_are_agreed),
        actual_working_days_are_agreed,
        week_including_day_contained_no_working_days_for_employee(D),
        wednesday(D).

qualifying_day_by_regulations(D) :-
```

```
        not(contractual_working_days_are_agreed),
        not(actual_working_days_are_agreed),
        not(day_on_which_no_employees_required_to_work(D)).

qualifying_day_by_regulations(D) :-
        not(contractual_working_days_are_agreed),
        not(actual_working_days_are_agreed),
        day_on_which_no_employees_required_to_work(D),
        week_including_day_contains_no_days_of_required_work_for_all_employees(D).

        %
        %       APPLICATION
        %

week_including_day_contains_no_days_of_required_work_for_all_employees(Day):-
        period_including_day(Week, week, Day),
        general_work_days_in_week(week, Work_days).

day_on_which_no_employees_required_to_work(Day) :-
        period_including_day(Week, week, Day),
        general_work_days_in_week(week, Work_days),
        in_period(Work_days, Day).

%
%       PAY DAYS
%
%       It is necessary to define separately 'pay_days' and 'days of
%       payment'. The two terms are used distinctly, the former are
%       used to define a period over which earnings are to be calculated,
%       the later to determine earnings within the period.
%

period_type(pay_day, event).
period_definition(pay_day, pay_day).

pay_day(D) :-
```

```
        employee_has_identifiable_normal_pay_days,
        normal_pay_day(D),
        day_of_payment(D).

pay_day(D) :-
        not(employee_has_identifiable_normal_pay_days),
        day_of_payment(D).

period_type(day_of_payment, event).
period_definition(day_of_payment, day_of_payment).

day_of_payment(D) :-
        pay_history(D, Sum_received, Start_pay_period, End_pay_period).
```

```
% ************************************************************
%
%       The following section contains procedures that take
%       a period definition and use it to find an instance of
%       that period subject to some constraint e.g. that it
%       must include a specific day, or be within a specific
%       time frame.
%
% ************************************************************

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Search for a period of general definition surrounding a day
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

period_including_day(P, Period_name, Day) :-
        pid(P1, Period_name, Day),
        first_day_conditions_pid(P_fd, P1, Period_name),
        excluded_pid(P_excl, P_fd, Period_name),
        valid_length_pid(P_len, P_excl, Period_name),
        special_conditions_pid(P, P_len, Period_name).

%
%       A linked period is broken down into its constituent parts first
%
pid(P, Period_name, Day) :-
        period_type(Period_name, linked),
        period_sub_period_name(Period_name, Sub_name),
        period_linkage(Period_name, Gap),
        period_including_day(Part, Sub_name, Day),
        !,
        linked_period(Part, Sub_name, Gap, P),
        !.

        %
```

```
        %       Boundary condition for assembling null part
        %
linked_period([ ],_,_,[ ]) :- !.

        %
        %       Assemble full period from the forward and backward linked
        %       sub parts
        %
linked_period(Part, Period_name, Gap, Full) :-
        linked_forward_period(Forward, Part, Period_name, Gap, _, _),
        linked_backward_period(Full, Forward, Period_name, Gap, _, _).

linked_forward_period(P, [Q|T], Period_name, Gap, _, _) :-
        !,
        end_list([Q|T], End),
        Search_start is End+2,
        Search_end is End+Gap+1,
        linked_forward_period(R, [ ], Period_name, Gap, Search_start, Search_end),
        append([Q|T], R, P).

linked_forward_period(P, P, Period_name, Gap, Search_start, Search_end) :-
        Search_start > Search_end,
        !.

linked_forward_period(P, [ ], Period_name, Gap, Search_start, Search_end) :-
        !,
        period_including_day(R, Period_name, Search_start),
        S is Search_start+1,
        linked_forward_period(P, R, Period_name, Gap, S, Search_end).


linked_backward_period(P, [Start|Q], Period_name, Gap, _, _) :-
        !,
        Search_start is Start-Gap-1,
        Search_end is Start-1,
        linked_backward_period(R, [ ], Period_name, Gap, Search_start, Search_end),
```

```
        append(R, [Start|Q], P).

linked_backward_period(P, P, Period_name, Gap, Search_start, Search_end) :-
        Search_start > Search_end,
        !.

linked_backward_period(P, [ ], Period_name, Gap, Search_start, Search_end) :-
        !,
        period_including_day(S, Period_name, Search_end),
        S is Search_end-1, .
        linked_backward_period(P, R, Period_name, Gap, Search_start, S).


%
%       Period type "framing" including particular day
%
pid(P, Period_name, Day) :-
        period_type(Period_name, framing),
        !,
        period_definition(Period_name, Def),
        day_in_period_backward(D, 0, Day),
        Pred =.. [Def,D],
        call(Pred),
        !,                                  %found the first day
        period_length(Period_name, L),
        construct_period_known_length(P, Day, L).


%
%       Period type "primitive_condition" including day
%
pid(P, Period_name, Day) :-
        period_type(Period_name, primitive_condition),
        !,
        period_definition(Period_name, Def),
        pid_prim(P, Def, Day).
```

```
pid_prim([ ], Def, Day) :-
        Condition =.. [Def,Day],
        not(call(Condition)),
        !.                                  % The day in question not in period

pid_prim(P, Def, Day) :-
        day_backward(D1, Day),
        Condition1 =.. [Def, D1],
        not(call(Condition1)),
        !,                                  % Found earlier limit
        day_forward(D2, Day),
        Condition2 =.. [Def,D2],
        not(call(Condition2)),
        !,                                  % Found later limit
        Start is D1+1,
        Finish is D2-1,
        Start =< Finish,
        construct_period_from_limits(P, Start, Finish).


%
%       Period type "primitive_event" including day
%
pid(P, Period_name, Day) :-
        period_type(Period_name, primitive_event),
        period_definition(Period_name, Definition),
        pid_prim_event(P, Definition, Day).

pid_prim_event(P, Def, Day) :-
        Event =.. [Def, Start, Finish],
        call(Event),
        Day >= Start,
        Day =< Finish,
        !,
        construct_period_from_limits(P, Start, Finish).
```

```
pid_prim_event([ ], _, _).                    % default.


%
%       Period type "derived" including day
%
pid(P, Period_name, Day) :-
        period_type(Period_name, derived),
        period_derivations(Period_name, [P_name|Derivations]),
        period_including_day(P1, P_name, Day),
        pid_derived(P, P1, Derivations, Day).

pid_derived(P, P, [ ], Day).

pid_derived(P_der, P, [Period_name|Derivations], Day) :-
        period_including_day(Q, Period_name, Day),
        intersection(P, Q, R),
        pid_derived(P_der, R, Derivations, Day).


%
%       Period type "event" including day
%       An event which occupies only one day is treated as a
%       degenerate form of period so that it can be handled
%       uniformly by other procedures.
%

pid(P, Period_name, Day) :-
        period_type(Period_name, event),
        period_definition(Period_name, Definition),
        pid_event(P, Definition, Day).

pid_event([Day], Definition, Day) :-
        Event =.. [Definition, Day],
        call(Event),
```

```
        !.

pid_event([ ], _, _).                   % Event did not occur on day.


%
%
%       The following definitions deal with the exclusions and special
%       conditions that can shorten one of the basic types.
%
        %
        %       Conditions that must be true of first day
        %
first_day_conditions_pid([ ], [ ], _) :- !.

first_day_conditions_pid(P, P, Period_name) :-
        not( period_first_day_conditions(Period_name, _)),
        !.

first_day_conditions_pid(P_id, P, Period_name) :-
        period_first_day_conditions(Period_name, Conditions),
        fdc_pid(P_id, P, Conditions).

fdc_pid(P, P, [ ]) :- !.

fdc_pid(P_id, [First_day|T], [C|Conditions]) :-
        Pred =.. [C, First_day],
        call(Pred),
        !,
        fdc_pid(P_id, [First_day|T], Conditions).

fdc_pid([ ], _, _).                     % Any failure suppresses period

        %
        %       Exclusive periods that suppress or terminate period early
        %
```

```
excluded_pid([ ], [ ], _) :- !.

excluded_pid(P, P, Period_name) :-
        not(period_exclusions(Period_name, _)),
        !.

excluded_pid(P_excl, P, Period_name) :-
        period_exclusions(Period_name, Exclusions),
        exc_pid(P_excl, P, Exclusions).

exc_pid([ ], [ ], _) :- !.

exc_pid(P, P, [ ]) :- !.

exc_pid(P_excl, P, [Period_name|Exclusions]) :-
        time_frame(P, Start, End),
        period_within_time_frame_forward(Excl, Period_name, Start, End),
        truncate_period(P, Excl, Trunc_P),
        exc_pid(P_excl, Trunc_P, Exclusions).

truncate_period(P, [ ], P) :-!.

truncate_period([S|X], [E|Y], [ ]) :-
        S > E,
        !.

truncate_period([S|X], [E|Y], [S|Z]) :-
        S < E,
        truncate_period(X, [E|Y], Z).


        %
        %       Period definitions may include ad hoc special
        %       conditions to deal with individual peculiarities
        %
```

```
special_conditions_pid(P, P, Period_name) :-
        not(period_special_condition(Period_name, _)),
        !.

special_conditions_pid(P_cond, P, Period_name) :-
        period_special_condition(Period_name, Condition),
        Pred =.. [Condition, P_cond, P],
        call(Pred).


        %
        %       length validity check
        %
valid_length_pid(P_len, P, Period_name) :-
        up_to_min_length_pid(P1, P, Period_name),
        within_max_length_pid(P_len, P1, Period_name).

up_to_min_length_pid(P, P, Period_name) :-
        not(period_min_length(Period_name, _)).

up_to_min_length_pid(P_len, P, Period_name) :-
        period_min_length(Period_name, Min),
        length_of_period(P, length),
        valid_length(P_len, P, Length, Min).

within_max_length_pid(P, P, Period_name) :-
        not(period_max_length(Period_name, _)).

within_max_length_pid(P_len, P, Period_name) :-
        period_max_length(Period_name, Max),
        length_of_period(P, length),
        valid_length(P_len, P, Max, Length).

valid_length([ ], P, X, Y) :-
        X < Y


valid_length(P, P, X, Y) :-
        X >= Y.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Search for a period within a specific time frame
%       Search forwards.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

period_within_time_frame_forward(P, Period_name, T_start, T_end) :-
        construct_period_from_limits(Frame, T_start, T_end),
        ptff(P1, Period_name, T_start, T_end).


ptff([ ], _, T_start, T_end) :-
        T_start > T_end,                % Solution is [ ] when none found
        !.

ptff(P, Period_name, T_start, T_end) :-
        period_including_day(P1, Period_name, T_start),
        ptff_next(P, P1, Period_name, T_start, T_end).

        % continue search if last clause did not find a solution %

ptff_next(P, [ ], Period_name, T_start, T_end) :-
        !,
        S is T_start+1,
        ptff(P, Period_name, S, T_end).

        % eject a solution %

ptff_next(P, P, _, _, _).

        % search for next solution %

ptff_next(P, Q, Period_name, _, T_end) :-
        end_list(Q, End),
        T_start is End+1,
```

```
        ptff(P, Period_name, T_start, T_end).

%       Search for a period of general definition within a specitic time frame.
%       Search backwards.
%
period_within_time_frame_backward(P, Period_name, T_start, T_end) :-
        construct_period_from_limits(Frame, T_start, T_end),
        ptfb(P1, Period_name, T_start, T_end),
        intersection(P1, Frame, P).

ptfb([ ], _, T_start, T_end) :-
        T_start > T_end,                % Solution is [ ] when none found
        !.

ptfb(P, Period_name, T_start, T_end) :-
        period_including_day(P1, Period_name, T_end),
        ptfb_next(P, P1, Period_name, T_start, T_end).

        % continue search if last clause did not find a solution %

ptfb_next(P, [ ], Period_name, T_start, T_end) :-
        !,
        E is T_end-1,
        ptfb(P, Period_name, T_start, E).

        % eject a solution %

ptfb_next(P, P, _, _, _).

        % search for next solution %

ptfb_next(P, [S|_], Period_name, T_start, _) :-
        T_end is S-1,
        ptfb(P, Period_name, T_start, T_end).
%
%       Accumulate all the periods of general definition within a specific
```

```
%       time frame into a single concatenated period.
%       Ordered in forward order.
%

all_periods_within_time_frame_forward(P, Period_name, T_start, T_end) :-
        construct_period_from_limits(Frame, T_start, T_end),
        aptff(P1, [ ], Period_name, T_start, T_end),
        intersection(P1, Frame, P).

aptff(P, P, _, T_start, T_end) :-
        T_start > T_end,                % Terminate the search
        !.

aptff(All, P1, Period_name, T_start, T_end) :-
        period_including_day(P2, Period_name, T_start),
        append(P1, P2, P3),
        aptff_next(All, P3, Period_name, T_start, T_end).


        % search for next solution %
aptff_next(All, [ ], Period_name, T_start, T_end) :-
        !,
        S is T_start+1,
        aptff(All, [ ], Period_name, S, T_end).

aptff_next(P, Q, Period_name, S, T_end) :-
        S1 is S+1,
        end_list(Q, End),
        S2 is End+1,
        max(S1, S2, T_start),
        aptff(P, Q, Period_name, T_start, T_end).
```

```
% ************************************************************
%
%       The following section contains miscellaneous
%       procedures that can perform useful operations on
%       time periods and days.
%
% ************************************************************

        %
        %       Generate days forward/backward from a reference
        %
day_forward(S, S).

day_forward(Day, Start) :-
        day_forward(D, Start),
        Day is D+1.

day_backward(F, F).

day_backward(Day, Finish) :-
        day_backward(D, Finish),
        Day is D-1.

        %
        %       Generate days forward/backward within a period
        %
day_in_period_forward(S, S, F) :-
        S =< F.

day_in_period_forward(Day, Start, Finish) :-
        Start < Finish,
        S is Start +1,
        day_in_period_forward(Day, S, Finish).

day_in_period_backward(F, S, F) :-
        S =< F.
```

```
day_in_period_backward(Day, Start, Finish) :-
        Start < Finish,
        F is Finish-1,
        day_in_period_backward(Day, Start, F).


        %
        %       Pick ctl first and last elements of list
        %
time_frame([Start|T], Start, Finish) :-
        end_list([Start|T], Finish).

end_list([X], X).

end_list([_|Y], Z) :-
        end_list(Y, Z).

        %
        %       Construct period as a list from known start and length/end
        %
construct_period_known_length([S], S, 1) :- !.

construct_period_known_length([Start|T], Start, Length) :-
        S is Start+1,
        M is Length-1,
        construct_period_known_length(T, S, M).


construct_period_from_limits([S], S, S) :- !.

construct_period_from_limits([Start|T], Start, Finish) :-
        S is Start+1,
        construct_period_from_limits(T, S, Finish).

        %
```

```
        %       intersection of ordered lists (periods)
        %
intersection([ ], _, [ ]).

intersection(_, [ ], [ ]).

intersection([X|Y], [X|Z], [X|S]) :-
        intersection(Y, Z, S).

intersection([X|Y], [P|Q], Z) :-
        X < P,
        intersection(Y, [P|Q], Z).

intersection([X|Y], [P|Q], Z) :-
        X > P,
        intersection([X|Y], Q, Z).


        %
        % list (period) append
        %
append([ ], L, L).
append([X|L1], L2, [X|L3]) :-
        append(L1, L2, L3).


        %
        %       Length of period (not number of elements in list,
        %       but distance between ends).
        %
length_of_period([ ], 0) :- !.

length_of_period(P, L) :-
        time_frame(P, S, E),
        L is E-S+1.
```

```
        %
        %       Number of days in period (number of elements in list)
        %

number_of_days_in_period([ ], 0).

number_of_days_in_period([X|Y], N) :-
        number_of_days_in_period(Y, M),
        N is M+1.

        %
        %       gap_between_periods
        %

gap_between_periods(P1, [E|P2], G) :-
        end_list(P1, S),
        G is E-S-1.

        %
        %       maximum of two numbers
        %

max(X, Y, X) :-
        X >= Y.

max(X, Y, Y) :-
        X < Y.


        %
        %       member of period
        %

in_period([P|_], P).

in_period([_|P], P) :-
```

```
        in_period(P, D).

        %
        %       Position of Day in period
        %

position_in_period([First|Rest], Day, _) :-
        Day < First,
        !, fail.

position_in_period([ ], Day, _) :-
        !,
        fail.


position_in_period([Day|T], Day, 1).

position_in_period([H|T], Day, N) :-
        position_in_period(T, Day, M),
        N is M+1.

        %
        %       Days of week
        %
        %       This is a simplistic implementation that assumes a
        %       reference sunday has been entered with problem data.
        %
ref_sunday(2).          % correct for 1983 which is used for the examples.

sunday(Day) :-
        ref_sunday(S),
        0 is (Day-S) mod 7.

monday(Day) :-
        ref_sunday(S),
        1 is (Day-S) mod 7.
```

```
tuesday(Day) :-
        ref_sunday(S),
        2 is (Day-S) mod 7.

wednesday(Day) :-
        ref_sunday(S),
        3 is (Day-S) mod 7.

thursday(Day) :-
        ref_sunday(S),
        4 is (Day-S) mod 7.

friday(Day) :-
        ref_sunday(S),
        5 is (Day-S) mod 7.

saturday(Day) :-
        ref_sunday(S),
        6 is (Day-S) mod 7.
```

```
% ***************************************************************
%
%       Listed here are all the bottom level goals that
%       can only be satisfied by external data.
%
% ***************************************************************

actual_working_day(D).
actual_working_days_are_agreed.
age(Date, Age).
agreed_qualifying_day(D).
contract_period(Start, Finish).
contractual_remuneration_is_multiple_month_entitlement(M, Salary).
contractual_remuneration_is_multiple_month_entitlement(M, Salary).
contractual_remuneration_is_weekly_entitlement(E).
contractual_working_day(D).
contractual_working_days_are_agreed.
day_of_sickness(Day).
employee_has_identifiable_normal_pay_days.
employer_exercises_right_to_withhold_payment_for_failure_to_notify(Day).
expected_confinement(Date).
general_working_days_in_week(Week, Work_days).
legal_custody(Start, End).
normal_pay_day(D).
notification_of_sickness_given(Day).
obligatory_qualifying_day_of_week(D).
pay_history(Pay_day, Sum, Start_pay_period, End_pay_period).
qualifying_days_are_agreed.
state_benefit(Start_period, End_period).
sunday(d).
```

```
%
%       CASE HISTORY 1
%

age(0, 27).

contract_period(0, 365).                % no contract dates are stated

qualifying_days_are_agreed.

qualifying_day(D) :-
        monday(D);
        tuesday(D);
        wednesday(D);
        thursday(D);
        friday(D).

employee_has_identifiable_normal_pay_days.

normal_pay_day(D) :-
        thursday(D).

pay_history(D, 150, D1, D2) :-
        thursday(D),
        D1 is D - 10,
        D2 is D -6.

day_of_sickness(D) :-
        D >= 310,
        D =< 317.

notification_of_sickness_given(D) :-
        D >= 312,
        D =< 317.

employer_exercises_right_to_withhold_payment_for_failure_to_notify(_).
```

```
%
%       CASE HISTORY 1
%
%       System response to problem queries
%
```

```
?- weekly_rate_of_ssp(310, Weekly_rate).

Weekly_rate = 37

?- period_within_time_frame_forward(SSPeriod, ssp_period, 310, 320).

SSPeriod = [315]

?- daily_rate_of_ssp(310, Daily_rate).

Daily_rate = 7.399994

?- ssp_due_in_period(310, 320, SSPeriod, SSPay).

SSPeriod = [315]
SSPay = 7.399994
```

```
%
%       CASE HISTORY 2
%

age(0, 42).

contract_period(0, 3o5).

day_of_sickness(D) :-
        D >= 224,
        D =< 231.

notification_of_sickness_given(L) :-
        day_of_sickness(D).

state_benefit(0, 180).
```

```
%
%       CASE HISTORY 2
%
%       System response to problem queries
%
```

```
?- period_within_time_frame_forward(SSPeriod, ssp_period,
        220, 240).

SSPeriod = [ ]

?- period_within_time_frame_forward(PIW, piw, 220, 240).

PIW = [224,225,226,227,228,229,230,231]

?- period_within_time_frame_forward(ENT, entitlement, 220, 240).

ENT = [ ]
```

```
%
%       CASE HISTORY 3
%

age(0, 27).

contract_period(C, 365).

actual_working_days_are_agreed.

actual_working_day(D) :-
        wednesday(D);
        thursday(D);
        friday(D).

pay_history(D, 47, D1, D) :-
        normal_pay_day(D),
        D1 is D-4.

employee_has_identifiable_normal_pay_days.

normal_pay_day(D) :-
        friday(D).

contractual_remuneration_is_weekly_entitlement(47.5).


day_of_sickness(D) :-
        D >= 120,
        D =< 132.

notification_of_sickness_given(D) :-
        D >= 120,
        D =< 132.
```

```
%
%       CASE HISTORY 3
%
%       System response to problem queries
%

?- all_periods_within_time_frame_forward(Qualifying_days,
        qualifying_day, 120, 135).

Qualifying_days = [124,125,126,131,132,133]

?- number_of_qualifying_days_in_week(120, N).

N = 3

?- weekly_rate_of_ssp(120, Weekly_rate).

Weekly_rate = 31

?- daily_rate_of_ssp(120, Daily_rate).

Daily_rate = 10.333328

?- ssp_due_in_period(120, 135, SSPeriod, SSPay).

SSPeriod = [131,132]
SSPay = 20.666626
```

```
%
%       CASE HISTORY 4
%

age(164, 65).

contract_period(0, 365).

qualifying_days_are_agreed.

qualifying_day(D) :-
        monday(D);
        tuesday(D);
        wednesday(D);
        thursday(D);
        friday(D).

employee_has_identifiable_normal_pay_days.

normal_pay_day(D) :-
        friday(D).

contractual_remuneration_is_weekly_entitlement(125).

pay_history(138, 250, _, _).            % Bonus payment

pay_history(D, 125, D1, D) :-
        normal_pay_day(D),
        D1 is D-4.

day_of_sickness(D) :-
        D >= 155,
        D =< 158.

day_of_sickness(D) :-
        D >= 166,
```

```
        D =< 168.

day_of_sickness(D) :-
        D >= 171,
        D =< 180.

notification_of_sickness_given(D) :-
        day_of_sickness(D).
```

```
%
%       CASE HISTORY 4
%
%       System response to problem queries
%
```

```
?- period_within_time_frame_forward(Sickness,
        sickness, 150, 180).

Sickness = [155,156,157,158]  y
Sickness = [166,167,168]  y
Sickness = [171,172,173,174,175,176,177,178,179,180]  y
Sickness = []

?- period_within_time_frame_forward(SSPeriod,
        ssp_period, 150, 180).

SSPeriod = [172,173,174,175,178,179,180]

?- ssp_due_in_period(150, 180, SSPeriod, SSPay).

SSPeriod = [172,173,174,175,178,179,180]
SSPay = 51.799561

?- normal_weekly_earnings(172, Weekly_earnings).

Weekly_earnings = 156.25

?- weekly_rate_of_ssp(172, Weekly_rate).

Weekly_rate = 37

?- daily_rate_of_ssp(172, Daily_rate).

Daily_rate = 7.399994
```

```
? ssp_due_in_period(171, 175, SSPeriod, SSPay).

SSPeriod = [172,173,174,175]
SSPay = 29.599854
yes
```

A N N E X

Appendix F
Examples of operation of SSP

1. This appendix gives 3 short examples and one longer one of spells of sickness for which action has to be taken under the SSP scheme.

If you want to check whether you have understood the operation of the SSP scheme correctly, you can look at these examples and work out the answers to the questions they ask.

The earnings levels and rates of SSP used are the ones given in this guide, *not* the ones which come into effect on 6 April 1983.

2. Example 1

A male employee aged 27 works 5 days a week and his agreed qualifying days are Monday to Friday inclusive. Your rules about notifying sick absence are that employees must telephone on the first qualifying day of sick absence.

He has not been sick in the past year and there are no unusual circumstances such as a stoppage of work.

You pay him full net pay under your own sick pay scheme for up to 4 weeks of sick absence each year, from the first day of sickness.

He earns a regular gross wage of £150 a week, payable on Thursdays.

He does not turn up for work on Monday 7 November 1983 and on Tuesday 8 November he telephones to say he is ill.

He returns to work on Monday 14 November and completes a self-certificate saying he was sick from Sunday 6 November to Sunday 13 November inclusive. He has no good reason for not telephoning you on Monday 7 November, but you have no reason to doubt that he was sick.

Questions

What is the appropriate weekly rate of SSP?
What is the appropriate daily rate of SSP?
What can you do about the late notification?
What SSP is due?
What do *you* have to pay?

Answers

Rate of SSP
The payments made to the employee over the period 9 September to 3 November average well over £60 a week, so the standard rate of SSP—£37 a week—is payable.

The daily rate is the weekly rate divided by the number of qualifying days in the week. In this case, £37 ÷ 5 = £7.40 per day.

Late notification
Notification was one qualifying day late with no good reason, *so if you wish* you can withhold payment of SSP for one day.

SSP due
The first 3 qualifying days, 7, 8 and 9 November, are waiting days for which SSP is not payable.

If you have decided to withhold payment of one day's SSP as a penalty for late notification of sick absence, you can withhold payment for Thursday 10 November. This would leave SSP only being payable for Friday 11 November and a total of £7.40 SSP would be due.

If you decided *not* to withhold any SSP, despite the late notification, SSP would be payable for both Thursday 10 November and Friday 11 November, a total of £14.80 SSP.

What do *you* have to pay?
If you are already paying this employee £150 in respect of these 5 days of absence under your own sick pay scheme, this comes to £30 a day, well in excess of your SSP liability for each payable day. You need pay no more than this. Your SSP liability has been met by your own sick payments, but you can, of course, get back the gross amount of SSP due.

3. Example 2

A female employee aged 42 is sick for a week. Only 6 weeks earlier she had been off sick and at that time she was claiming State invalidity benefit and was not receiving SSP.

She states on a self-certificate that her first day of sickness was 10 August 1983. You hold a letter from the DHSS stating that spells of sickness starting on or before 23 August 1983 should result in claims for State benefit rather than SSP.

Questions

What action, if any, should you take under the SSP scheme? When should you take action?

Answers

The sickness lasted more than 3 days, so a PIW has been formed: but this employee is *excluded* from SSP.

You should send or give her an exclusion form SSP 1(E) within 7 days of knowing she had been sick for at least 4 days.

You should record the sickness absence, the fact that the employee was excluded from SSP, and the reason for the exclusion.

4. Example 3

A female employee, aged 27, works for you in the mornings only, each Wednesday, Thursday and Friday. You have not agreed any qualifying days with her, but she is aware of your rules about notification of sickness which say that employees should telephone you on the first qualifying day of sick absence.

She does not come in to work on Wednesday 4 May 1983 but telephones on the same day to say she is ill. It is her first sick absence for over a year. She is *not* entitled to any occupational sick pay from you. On Monday 9 May you receive a self-certificate form stating that she has been incapable of work since Saturday 30 April, together with a doctor's statement dated 7 May which states that she will be fit to return to work on Friday 13 May. She does return to work on that day.

There is no reason to doubt her incapacity, and no reason to suspect that she might be pregnant.

Her wages over the 8 weeks before she fell sick have been a regular £47.50 a week gross, payable on Fridays.

Questions

Which days are qualifying days?
How many qualifying days in a week?
What is the appropriate weekly rate of SSP?
What is the appropriate daily rate of SSP?
What payment of SSP should be made?

Answers

Qualifying days
In the absence of an agreement as to qualifying days, they are those days which you and your employee agree would normally have been worked. In this case the qualifying days are Wednesday, Thursday and Friday of each week.

There are 3 qualifying days in each week.

Weekly rate of SSP
All payments made in the period 5 March to 29 April should be added together and divided by 8. The result in this case is average weekly earnings of £47.50. Thus the appropriate weekly rate of SSP is £31 (middle rate).

Daily rate of SSP
The daily rate is the weekly rate divided by the number of qualifying days in the week.

£31 ÷ 3 = £10.3333 a day.

SSP payable

The spell of sickness included 5 qualifying days. The first 3 are waiting days and SSP is payable only for 11 and 12 May.

The SSP payable is $\frac{2}{3}$ of £31 which = £20.6667. This is rounded up to the next whole penny to give a gross payment of £20.67 SSP.

5. Example 4

A long-serving male employee aged 64 goes sick. His agreed qualifying days are Monday to Friday inclusive each week. He has had no previous sick absence for 3 months.

Your rules about notification of sick absence are that employees should telephone you on the first qualifying day of sick absence.

He telephones on Monday 6 June 1983 to say he is not well, and will be away for a few days. He returns to work on Wednesday 8 June, when he completes a self-certificate saying he had been ill from Saturday 4 June to Tuesday 7 June inclusive. There is no reason to doubt this statement.

Monday of the next week, 13 June, is his 65th birthday. He is away sick on the Wednesday, Thursday and Friday of that week (he telephones on the Wednesday to report his absence). He posts to you a self-certificate stating that he was sick for these 3 days only, and was *not* incapable of work on Saturday 18 June or Sunday 19 June.

However, he does not return to work on Monday 20 June. Evidently he has become seriously ill. His daughter telephones on Wednesday 22 June to say that he was taken to hospital on Monday and you eventually receive a hospital certificate (form Med 10) which confirms this.

He returns to work on 20 September 1983.

There are no unusual circumstances such as strikes at his place of work during his sickness.

He is normally paid weekly on Friday for the week up to and including that Friday. On each of the 8 pay days up to and including Friday 3 June he received £125.00 gross. On Wednesday 18 May he received a bonus payment of £250.

Questions

1. There are 3 spells of sickness. What SSP action do you take in respect of each one?

2. What are the employee's average weekly earnings for SSP purposes?

3. What are the appropriate weekly and daily rates of SSP?

4. Assuming that you wish to pay any SSP due on a weekly basis, how much SSP would be due on each Friday while the sickness lasts?

5. When should a transfer form be sent to the employee?

Answers

1 First spell
The employee was sick from 4–7 June inclusive. This is 4 days, so a PIW has been formed.

The employee is not excluded from SSP.

There are 2 qualifying days in the PIW, which will be *waiting days*.

No SSP is payable in respect of this spell and, having noted your records, there is no further action you need to take.

Second spell
The employee was sick from 15–17 June inclusive. This is less than 4 consecutive days. A PIW has *not* been formed and you should take no action at all under the SSP scheme.

Third spell
The employee was sick from 20 June to 19 September, so a PIW has been formed.

This PIW *links* with the *first* PIW. There must be 14 or fewer days between 2 PIWs to establish a link. From 8 June to 19 June (inclusive) is 12 days, so the 2 PIWs are linked. (The second spell, which did *not* form a PIW, is ignored).

So the 2 PIWs count as one.

The employee reached pension age on 13 June and if the 3rd spell had not linked with an earlier spell, he would have been excluded from SSP for this reason. However, the first day of sickness in the series of linked PIWs was 4 June, *before* he reached pension age. So he is not excluded from SSP.

You were notified late in respect of 2 qualifying days at the beginning of the 3rd spell of sickness, but as the employee had been taken into hospital there was a good reason for the delay and no penalty is imposed.

Two waiting days were served in the first linked PIW. One more waiting day remains to be served, so SSP is due from the second qualifying day of the 3rd spell of sickness; ie from Tuesday 21 June.

2. The employee's average weekly earnings are calculated by adding together his gross earnings over the period 9 April to 3 June inclusive. The bonus received on 18 May must be included. Thus (8 × £125 + £250) ÷ 8 = £156.25.

3. The weekly rate of SSP is therefore £37. The daily rate is the weekly rate divided by the number of qualifying days in the week. £37 ÷ 5 = £7.40 per day.

4. On Friday 24 June, 4 days' SSP are due. 4 days at £7.40 = £29.60.

A full week's SSP, £37, is due on each of the next 7 Fridays: 1, 8, 15, 22 and 29 July and 5 and 12 August.

A single day's SSP, £7.40 in respect of Monday 15 August, is due on Friday 19 August.

5. You should issue a transfer form on any convenient day between Tuesday 2 August and Tuesday 9 August; ie between the day on which there are 2 weeks' more SSP due, and the day on which one more week of SSP is due.