Mehldau, Gerhard

A RULE-BASED PROGRAMMING LANGUAGE AND ITS APPLICATION TO IMAGE RECOGNITION

The University of Arizona M.S. 1986

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark __✓__.

1. Glossy photographs or pages __✓__
2. Colored illustrations, paper or print __✓__
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages _____
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) _____ lacking when material received, and not available from school or author.
12. Page(s) _____ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Dissertation contains pages with print at a slant, filmed as received _____
16. Other _____

_____

_____

University
Microfilms
International

A RULE-BASED PROGRAMMING LANGUAGE AND ITS APPLICATION TO IMAGE RECOGNITION

by

Gerhard Mehldau

A Thesis Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE

In the Graduate College

THE UNIVERSITY OF ARIZONA

1986

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission of extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Gerhard Fehlde

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

E. W. MYERS
Assistant Professor of Computer Science

June 18, 1986
Date

Table of Contents

List of Illustrations

List of Tables

ABSTRACT

Most current systems for prototyping rule-based systems employ an applicative approach. They use techniques such as heuristic search and are generally written in LISP dialects. These design choices make it easy to prototype small applications in a flexible way. However, such systems are less useful in production applications involving large amounts of input data and application-dependent, numerically-oriented libraries. Most production environments center on languages like FORTRAN, PASCAL, or C and employ personnel inexperienced in AI techniques and LISP, further limiting the utility of current prototyping systems.

C is a language that is widely available and with which many programmers are familiar. An augmentation of C is presented containing the control structures, predicates, and data types that make possible the convenient specification of a rule-based system. A preprocessor translates the extended C software for a rule-based system into standard C software which is subsequently compiled. This makes the prototype more efficient than those that are interpreted. In addition, libraries of application-dependent software written in C can easily be integrated, and personnel need only learn the small number of concepts added to C.

The new features added to C are described in detail, together with their restrictions and limitations. The implementation of these features is discussed. The system is applied to build a rule-based system for labeling features in a digitized aerial photograph. Aspects of engineering the application and the performance of the resulting tool are discussed.

CHAPTER 1

RELATED WORK

Long before the first computer was built, man was fascinated by the idea of building machines that could think like human beings. The dream of intelligent machine behaviour seemed to be very close to realization when in 1946 the first general-purpose electronic digital computer became operational. As early as the 1950s, a whole branch of Computer Science dedicated itself to research in the field of Artificial Intelligence. But even though some spectacular results have been achieved, the progress made could not live up to the expectations.

A major research effort in the field was directed towards so-called "expert systems." Expert systems are programs that are able to perform a difficult task at a level that is comparable to a human expert. Examples of these tasks include problems from such specialized fields as symbolic mathematics (theorem proving), chemical analysis, medical diagnosis, and visual and speech perception.

In the specialized domains, research was much more successful than in the domain of the more general problems. For example, DENDRAL (Buchanan, Sutherland, and Feigenbaum, [2]) is an expert system developed at Stanford that infers the molecular structure of chemical compounds from

mass spectrographic data. MYCIN (Shortliffe, [23]) advises physicians on diagnosis and therapy for infectious diseases. Both programs perform at a level that compares to or even surpasses human experts in the field.

Rule-based systems are a special type of expert systems. They consist of a rule base (a set of rules about the task in general), a knowledge base that contains all the facts that are known about the specific problem at hand, and a search procedure. The search process can use either forward or backward chaining. Forward chaining applies the rules to the facts, producing conclusions in the hope of uncovering the desired solution. Backward reasoning starts with the goal and attempts to find a path through the rules that leads to some confirmed facts. Both techniques usually employ heuristics to limit the scope of the search process.

In the early days of expert systems, the writer of every new system had to start from scratch, but soon it was recognized that all the systems had a common structure. It was only a small step from there to the idea of writing programs that could automatically generate expert or rule-based systems. Most of the research that was done in AI used LISP or one of its dialects. For an introduction to the subject see Hayes-Roth, Waterman and Lenat [7]; for a survey of some current systems see Cross [4].

Systems for the automatic generation of rule-based systems can be divided into two distinct groups: systems or languages that were specifically developed for this purpose, and languages that were not, but can be used for generating rule-based systems. The latter group mostly consists of

languages for logic programming, such as Prolog (Clocksin and Mellish, [3]) or MRS (Russell, [20]).

MRS is a language for logic programming and reasoning that is embedded in LISP (Steele, [24]). In addition to a great number of relations and predicates, MRS also has some meta-level facilities. This means that a user of the system has (at least to some extent) control over the reasoning process. In MRS, the rule base and the knowledge base are one. Rules and data (knowledge) may be added to or deleted from a data base. Inference takes place in response to queries where the user asks the system to either ascertain a fact or, using the rules, to infer it from other facts. The inference process is controlled by the meta-level knowledge. Meta-level rules are a subset of the rule base and they determine whether forward or backward chaining is used, which rules are preferred over others, which rules are applicable when, and so on. Another convenience of MRS is the possible escape to the underlying LISP environment. Certain things are difficult to express or hopelessly inefficient in relational programming. In MRS, they can be written in procedural form in LISP and incorporated into the database.

One thing not built into MRS, but which is an important characteristic of expert systems, is the notion of uncertain knowledge. MRS only knows true and false, known and unknown, provable and unprovable. In the real world, however, little is one hundred percent true or false. Therefore, systems that are designed for the purpose of generating rule-based

systems, usually provide facilities for reasoning in the presence of uncertainty. Examples include systems such as EMYCIN (Van Nelle, [26]), KAS (Duda, Hart, Nilsson and Sutherland, [5]), and EXPERT (Weiss and Kulikowski, [28]).

All these systems associate certainty factors, or probabilities, with every fact. The reasoning process then provides mechanisms to combine these "degrees of truth" with conjunctions and disjunctions. In order for a predicate to be true in the conditional part of a rule, the resulting certainty must be above some threshold. The action part of a rule can either update probabilities of old facts or produce new facts and assign certainty factors to them.

All systems mentioned above have additional, sophisticated features that distinguish them from simpler languages and make them easier to use. These features include the possibility of explaining their reasoning, editors designed to operate on the knowledge base, and tools for tracing and debugging.

The system presented here does not attempt to compete with those systems in terms of generality or sophistication. Instead, the purpose of the new language is to provide a convenient means of specifying a system for rule-based classification. Furthermore, the system should fit well into production environments and be capable of handling and processing large amounts of data in a reasonably efficient manner.

CHAPTER 2

THE SYSTEM

Most of the systems that have been developed for automatically generating rule-based systems employ an applicative approach. They use AI-techniques such as heuristic search and backtracking and are generally written in LISP or LISP dialects. These design choices make it easy to quickly produce a working system and thus encourage experimenting with different approaches to the problem at hand.

However, for problems where large amounts of input data are involved, such systems are less useful. The code they produce is in most cases interpreted, not compiled, and therefore very slow. Furthermore, most conventional production environments for numerical computation are FORTRAN-based and in general do not have compilers or interpreters available for LISP. In cases where libraries of application-dependent software already exist, the interface to the rule-based system creates an additional problem. Last, but not least, programmers in production environments are usually unfamiliar with the concepts behind applicative languages and with their use.

These reasons initiated the development of a system that potentially overcomes these problems. The standard C language has been augmented

with new control structures, predicates, and data types to permit the convenient specification of a rule-based system. A system written in this extension of C is translated into standard C by a preprocessor and subsequently compiled. The advantages of this method are threefold:

- The compiled code is faster than interpreted systems.
- Existing C software for the application at hand can easily be integrated.
- Programmers need only become familiar with a few new concepts.

The overall purpose of the new language is to provide a system that will classify a sequence of input records into categories specified by rules. The extensions to C correspond to the basic constituents of any rule-based system. The concept of rules is added to C and a number of new predicates are provided to help in their specification. The search procedure is implemented with the concept of phases that iterate over the rules. A new data type, lists, is added for convenience. Features are provided to facilitate the input and output of data into and from the knowledge base.

The approach taken here is to assign to each input record a number of labels that represent the classification. The rules used in the classification process are grouped into phases, each of which determines a specific label. A rule consists of a condition and any number of consequences, one of which must be the assignment of a label.

Each phase assigns values to exactly one label. More than one value may be assigned to any label, since the data often permits multiple interpretations or classifications. This is accomplished by evaluating the rules in

one phase, at least conceptually, in parallel. If more than one rule in a given phase yields true for a given input record, a tree is created. Each node of the tree represents a value assigned to the label. The siblings of a node represent the possible values of a label.

The different phases are best thought of as levels in what will be called the "label tree." Therefore, this tree is built in a breadth-first manner, one level per phase. A first phase creates the top level of the tree. Subsequent phases evaluate their rules for each possible path through the tree, appending a new level to that path. When eventually all phases are evaluated, each path in the tree represents a complete classification.

Rules can refer to labels assigned in earlier phases in both their conditional and consequential parts and thus refine earlier classifications. It is also possible to assign the same label in more than one phase. This is useful if new information has been computed and old labels (or their probabilities) are no longer appropriate.

Confidence values or probabilities are an important concept associated with the classification. Parallel with the assignment of a label, a probability is assigned that represents the confidence in the particular label. Confidence values can be propagated down the tree to reflect connections between classifications.

In AI-terminology, this system uses forward-chaining. All possible classifications are enumerated indiscriminately until eventually a solution is found.

It is shown that the implementation of this system is feasible. An example of an application in the domain of image recognition is given and the results obtained are discussed.

Chapters 3 and 4 define the new features that have been added to C. In Chapter 3 the exact syntax and semantics of each feature is stated. Chapter 4 develops an extensive example to illustrate their use.

The implementation of the extensions is discussed in Chapters 5 to 8. Chapter 5 covers rule searching. Chapter 6 explains the evaluation of predicates. Chapter 7 discusses the implementation of lists, together with the restrictions and limitations imposed. Chapter 8 looks at the mechanisms provided for input and output.

An application of the system is presented in Chapters 9 to 12. Chapter 9 is a general introduction to the problem domain: labeling features in a digital aerial photograph. Chapter 10 covers the preprocessing steps necessary to prepare the image for classification and Chapter 11 deals with the rule-based interpretation. Results are presented and discussed in Chapter 12.

CHAPTER 3

SYNTAX AND SEMANTICS

In this chapter, the extensions to C are formally defined. The syntax of each new feature is specified with a modified BNF-grammar. A syntactic element being defined is listed, followed by a "$\leftarrow$" and its definition. Syntactic elements that are listed on one line need to appear in the same order in the program; alternatives are separated by "|". Normal type indicates terminals (e.g. keywords) that appear literally in the program. Other syntactic elements (i.e. non-terminals) are printed in italics. Syntactic elements that are not defined in this chapter are standard C constructs and described in Kernighan and Ritchie ([10], pp. 214-219). A "+" after a syntactic element indicates one or more occurences of this element, and a "*" means zero or more occurences. Syntactic elements enclosed in "<" and ">" can appear either zero or one times.

We separate the extensions to C into four classes. These four groups are, in order of presentation:

- List Data Type
- Input / Output
- Control Structures
- Expressions

For the elements in each class, we will describe the syntax and semantics and give a small example of their use. For a complete example of a rule-based system, see Chapter 4.

List Data Type

Lists have been added to C as a new data type to make possible the convenient handling of data sequences of variable length. Lists are declared with the "@"-sign in a syntax similar to the "*" or "pointer-to" notation. The "@" can appear both in type names and in variable declarations.

declarator ← @ *declarator*

The following are all list expressions which can appear anywhere in a program. Elements of a list are addressed like elements of an array:

element-selection ← *expression* [*expression*]

where the first expression must be a variable of type "list of ... ," and the second expression must evaluate to an integer in the range 0..(length of list)-1. The system does not check for indices that are out of range.

Sublists of lists are selected similarly:

sublist ← *expression* [*expression* .. *expression*]

Again, the first expression must be a variable of type "list of ... ," and the second and third expressions must evaluate to an integer in the range 0..(length of list)-1.

The length of a list can be determined using the len-function:

listlength ← len (*expression*)

where the expression in parenthesis must be a list.

Lists can be concatenated using the "*"-operator, which in C stands for multiplication when used with expressions of numeric type.

list-concatenation ← *expression* * *expression*

Both expressions must be of type list and their contents should be of the same type.

Lists are assigned to variables with two of the standard C assignment operators:

list-assignment ← *lvalue* = *expression*

| *lvalue* *= *expression*

All of the expressions involved must be lists. The first version of the list assignment assigns the value of the expression to the lvalue. The second version appends the second list to the end of the first.

Finally, there is a way to specify list constants:

list-constant ← [<*range-expression* (, *range-expression*)*>]

This says that a constant list is either empty or consists of a number of range-expressions which are separated by commas. A range-expression is either a single value or a range of values from the first to the second.

range-expression ← *expression*

| *expression* .. *expression* /* "in"-operator only */

An example of a function that makes use of lists is given in Figure 1. The argument to the function is a list of characters. The function builds and returns a new list that contains all lowercase characters from the argument list.

```
char @example (list) char @list;
{ int i;
  char @newlist;
  newlist = [];
  for (i = 0; i < len(list); i++)
    if ('a' <= list[i]  &&  list[i] <= 'z')
      { newlist *= list[i..i];
        /* The next line has the same effect as the last one. */
        /* newlist = newlist * [list[i]]; */
      }
  return (newlist);
}
```

Figure 1: Sample list function

Input / Output

Before any classifications can be attempted, our system needs to read the input data. It is assumed that the input consists of a list of uniform records. Therefore, the user need only specify the type of one of those records. This is done using the "input" construct, whose syntax follows:

input ← input { *field-declaration+* } <*declarator* (, *declarator*)*>

The field declarations inside the curly braces are the same that are used in

standard C for structures. The declarator(s), if present, name the input type just as if "input" were replaced with "typedef struct" in standard C. The input records are stored in the implicit system variable "INPUT." This variable can be accessed like any other variable of type list in the new language.

An example for an input declaration is given in Figure 2. Here, an individual record consists of an integer, "id", two real numbers, "a" and "b", and a list of characters, "string".

```
input {
  int id;
  double a, b;
  char @string;
} example;
```

Figure 2: Sample input declaration

The system provides the user with routines that will do the input and the output in a standard fashion.

For the input, a function will be generated that reads a list of the records described in the "input"-declaration from the standard input. The input records have to be in the same format that is used by C initializers for structures. The input syntax for lists is the same as for arrays. There are, however, some restrictions on the fields of the structure: it cannot contain any pointers, unions and enumerated types.

For the output, the system provides the user with a function that writes the output in a standardized form to the standard output. The input is echoed in a more readable form: brackets are omitted and each field of the input structure is printed on a separate line. The results of the classification process are appended to each record. Every label assigned is printed, together with its probability and the rule that generated it. The labels are indented so as to show the underlying tree.

Users familiar with the system have the option of writing their own output function. If the program specification supplied to the system contains a function definition for "output," the automatic generation of this function is suppressed.

Control Structures

The two control structures added to C are called phases and rules. A phase is integrated into the system as an alternate form of an external definition and contains a set of rules which produce one level of the label tree.

phase ← phase *identifier* (*identifier*) *declaration rule-block*

This definition asserts that a phase consists of the keyword "phase," followed by a name for the phase. The identifier in parentheses names the label that is assigned to the input structures during that phase, according to the rules specified in the phase. The label identifier is followed by its declaration. The body of a phase, called a rule-block, is defined as follows:

rule-block ← { *declaration** <*statement*> *rule-list* <*statement*> }

A rule-block consists of any number of declarations for local variables, an optional statement, a list of the rules that are being applied during the phase and possibly another statement; all enclosed in curly braces. The optional statements before and after the list of rules are executed only once by the program, but the rules are executed in a loop — conceptually one iteration per input record. A rule-list is a sequence of at least one rule:

rule-list ← *rule+*

A rule consists of the keyword "rule" and a name for the rule, followed by a conditional expression, the "implies"-operator "=>" and the consequences. The rule name appears in the output, which is automatically generated, so that the system's path of reasoning can be traced.

rule ← rule *identifier expression* => *consequences*

The conditional expression may be any (integer) C expression and can include the expressions that will be defined later in this chapter. The consequences consist of either a single label-assignment or of a block that contains a label-assignment:

consequences ← *label-assignment*

| { *declaration* statement* label-assignment statement** }

The purpose of the label-assignment is to assign a value to the label that has been declared together with the phase and to associate a "confidence value" or probability with it. The alternate form is used if, in addition to the label-assignment, other computations are desired in response to a "true" rule. The syntax for the label-assignment is:

label-assignment ← assign (*expression*) *expression* ;

This statement will assign the value of the second expression to the label. It is the user's responsibility to ensure that the type of the label and the type of the second expression are compatible. The expression in parentheses is the probability associated with the label and can be of any arithmetic type.

The sample phase shown in Figure 3 assumes the input configuration from the example in Figure 2. It obtains two additional values (presumably interactively), and assigns a label to the current record if the sequence "value_1, a, b, value_2" is either increasing or decreasing. The question mark in the code stands for a pointer to the current record. This notation is explained in the next section.

```
input { int id;  double a, b;  char @string; };

phase example (label) char *label;
{ int value_1, value_2;

  obtain_values(&value_1,&value_2);

  rule one
  value_1 < ?->a  &&  ?->a < ?->b  &&  ?->b < value_2  =>
    assign (1.0) "increasing sequence";
  rule two
  value_1 > ?->a  &&  ?->a > ?->b  &&  ?->b > value_2  =>
    assign (1.0) "decreasing sequence";

  printf("Phase finished\n");
}
```

Figure 3: Sample phase declaration

Expressions

To help the user of the system access the input data and express properties of objects, the syntax of standard C for expressions has been enhanced. However, not all of these expressions can be used everywhere. If there are any restrictions to the scope of a new expression, they are stated in this chapter.

To access the input data, the "question mark"-notation is used. A question mark ("?") can, in general, stand in any place within the list of rules where an identifier (variable) is required and it denotes a pointer to the record that is currently being processed.

current-record ← ?

Within rule-lists, the user can obtain the probability of all the labels assigned so far. This is done by means of the probability-"function":

probability ← prob (*expression*)

where the expression in parenthesis must be a reference to one of the labels that are defined at this point. The reference is always to the most recent assignment of the label.

There are three new expressions that help simplify the formulation of rule-predicates: the "forall"- and "exist"-quantifiers and the "in"-expression. Both quantifiers are restricted to the conditional part of rules. The syntax of the "forall"-quantifier is

forall ← for_all *identifier* in *expression* : *expression*

where the first expression must be of type list and the identifier must be a variable of the same type as the elements of the list. The expression after the colon is the condition and can again be any (integer) C expression. "Forall" evaluates to true if and only if the second expression is true for all elements of the list. A similar syntax is used for the "exist"-quantifier:

exist ← there_exists *identifier* in *expression* : *expression*

An "exist"-expression is true if the list contains at least one element for which the conditional expression holds. The precedence of the ":"-operator is lower than that of any operator in standard C.

The precedence of the "in"-operator is the same as for the comparison "equal" in standard C and its syntax is:

in ← *expression* in *expression*

The second expression must be a list constant (see above), and the first expression must be of the same type as the elements in that list. "In" evaluates to true if and only if the first expression is either equal to one of the constants in the list or, in the case of a range-constant, is within the specified range.

Figure 4 shows an example of a phase that uses the new expressions. Again we assume the input configuration from Figure 2. Both rules scan the character string of the current record to determine which label to assign.

```
input { int id;  double a, b;  char @string; };

phase example (label) char *label;
{ char c;

  rule one
  for_all c in ?->string : c in ['a'..'z'] =>
    assign (1.0) "lower case";
  rule two
  (for_all c in ?->string : c in ['a'..'z','A'..'Z']) &&
  (there_exists c in ?->string : c in ['a'..'z']) &&
  (there_exists c in ?->string : c in ['A'..'Z']) =>
    assign (1.0) "mixed case";
}
```

Figure 4: Sample phase declaration

CHAPTER 4

EXAMPLES

In this chapter, we present a small example of a rule-based system. The purpose of this program is to determine the names of animals, given a description of their properties. We assume that an input record (the description of an animal) consists of a unique ID and a list of properties. Integers were chosen over character strings to represent the properties, because integer comparisons are easier to express and faster to execute. We therefore start with the following program:

```
#define FEET        0
#define WINGS       1
#define FINS        2
#define FUR         3
#define CLAWS       4
#define HAPPY       5
#define UGLY        6
#define BLOWHOLE    7

input {
  int id;
  int @properties;
} whatzit;
```

In general, a program in the extended language consists of any number of external declarations for types, variables and functions as in standard C, but in addition contains an input specification and any number of phase

declarations. Since the main program will be generated by the system, there must not be a function of that name in the program. For the above declaration, our system will produce a complete C program which contains a variable declaration for a list of records containing an integer and a list of integers each, an input function that reads the records from the standard input, and an output function that writes the "processed" input and the results obtained to the standard output. It will also contain a main program that consists of calls to the input and output functions. The generated program therefore simply reads the input, reformats it, and prints it out.

As a next step we define a phase which assigns one of the general groups "mammal," "bird," or "fish" to an animal. To do this, we scan the property list of every animal for the characterisics "feet," "wings," and "fins," respectively. For each property found, we assign the corresponding label and a certainty value.

```
char *MAMMAL =    "Mammal",
     *BIRD =      "Bird",
     *FISH =      "Fish";

phase one (group) char *group;
{ int property;

 rule one_1
 there_exists property in ?->properties : property == FEET =>
  assign (0.5) MAMMAL;
 rule one_2
 there_exists property in ?->properties : property == FEET =>
  assign (0.5) BIRD;
 rule one_3
 there_exists property in ?->properties : property == WINGS =>
  assign (1.0) BIRD;
 rule one_4
 there_exists property in ?->properties : property == FINS =>
  assign (0.9) FISH;
 rule one_5
 there_exists property in ?->properties : property == FINS =>
  assign (0.1) MAMMAL;
}
```

In the second and last phase we refine the classification of the first phase by looking at both the "group"-label and at some of the original properties.

```
phase two (animal) char *animal;
{ int property;

 rule two_1
 ?->group == MAMMAL  &&
 (there_exists property in ?->properties : property == FUR)  &&
 (there_exists property in ?->properties : property == CLAWS)  =>
   { assign (prob(?->group)) "Wildcat";
     printf("Caution: Animal is dangerous!\n");
   }
 rule two_2
 ?->group == MAMMAL  &&
 there_exists property in ?->properties : property == HAPPY  =>
   assign (prob(?->group)) "Koala Bear";
 rule two_3
 ?->group == BIRD  &&
 there_exists property in ?->properties : property == UGLY  =>
   { assign (prob(?->group)) "Vulture";
     printf("Beware!\n");
   }
 rule two_4
 ?->group == BIRD  &&
 there_exists property in ?->properties : property == HAPPY  =>
   assign (prob(?->group)) "Nightingale";
 rule two_5
 ?->group == FISH  &&
 for_all property in ?->properties : property != BLOWHOLE  =>
   assign (prob(?->group)/10.0) "Goldfish";
 rule two_6
 ?->group == MAMMAL  &&
 there_exists property in ?->properties : property == BLOWHOLE  =>
   assign (prob(?->group)) "Whale";
}
```

For the sample input

{{1,{0,3,4}},{2,{0,5}},{3,{1,6}},{4,{2,4,6}},{5,{2,7}}}

our system produces the following output:

```
Caution: Animal is dangerous!
Beware!
==========================================
RECORD #1:
INPUT:
1
{ 0, 3, 4 }
OUTPUT:
  LABEL: Mammal   PROB: 0.500000   RULE: one_1
    LABEL: Wildcat   PROB: 0.500000   RULE: two_1
  LABEL: Bird   PROB: 0.500000   RULE: one_2
    NO LABEL ASSIGNED
==========================================
RECORD #2:
INPUT:
2
{ 0, 5 }
OUTPUT:
  LABEL: Mammal   PROB: 0.500000   RULE: one_1
    LABEL: Koala Bear   PROB: 0.500000   RULE: two_2
  LABEL: Bird   PROB: 0.500000   RULE: one_2
    LABEL: Nightingale   PROB: 0.500000   RULE: two_4
==========================================
```

```
RECORD #3:
INPUT:
3
{ 1, 6 }
OUTPUT:
  LABEL: Bird   PROB: 1.000000   RULE: one_3
    LABEL: Vulture   PROB: 1.000000   RULE: two_3
==========================================
RECORD #4:
INPUT:
4
{ 2, 4, 6 }
OUTPUT:
  LABEL: Fish   PROB: 0.900000   RULE: one_4
    LABEL: Goldfish   PROB: 0.090000   RULE: two_5
  LABEL: Mammal   PROB: 0.100000   RULE: one_5
    NO LABEL ASSIGNED
==========================================
RECORD #5:
INPUT:
5
{ 2, 7 }
OUTPUT:
  LABEL: Fish   PROB: 0.900000   RULE: one_4
    NO LABEL ASSIGNED
  LABEL: Mammal   PROB: 0.100000   RULE: one_5
    LABEL: Whale   PROB: 0.100000   RULE: two_6
==========================================
```

CHAPTER 5

RULE SEARCHING

The next four chapters cover the implementation of a preprocessor for the new language. In this chapter we will look at the control structures that govern the searching of rules. The basic approach is a breadth-first search of the label state space. We now examine the control structures at different levels in the system.

The preprocessor generates a main program that consists of only a few function calls: one call to the input function, then one call per phase (in the order of declaration) and a final call for the output.

For each phase, two functions are generated. The first one initializes a new label and appends it to the label tree of the record currently being processed. The tree is encoded in a son-brother fashion and its nodes contain an additional pointer to a data structure that is potentially different for each level of the tree (that is, for each phase). This data structure contains three fields: a rule name, a label value and its probability. The rule name contains the name of the rule that created the node, and the label value and probability are those assigned by the rule.

The second function implements the actual searching process. Its operation is best explained by looking at the template that is used for its

creation. For a phase that looks like

```
phase name (label) declaration;
{ declarations
  statement
  rules
  statement
}
```

we produce the following function:

```
name()
{ declarations
  statement
  for each input record do
    { for all possible paths through label tree
        { rules }
    }
  statement
}
```

For each record in the input list the program loops through the rules at least once. The actual number of iterations per record depends on the growing label tree. Since the rules can reference assigned labels, one iteration is necessary for each possible path through the label tree for each record. We maintain pointers, one per level of the tree, that show us the path currently being explored. The pointers are updated after each iteration. If a rule evaluates to true during the search, a new node is appended to that path, thus creating a new level in the tree. However, label values assigned in the current phase are not included in the iteration. It is important that each phase appends a label to the current path, since subsequent phases can refer to labels that were assigned earlier. If none of the rules apply to a potential

classification, then a dummy node is created and appended to the corresponding path to ensure its completeness.

The evaluation of the rules themselves is straightforward. The conditional part is transformed into an if-statement and evaluated as described in the next chapter. If true, a call occurs to the function that appends a node to the tree. Subsequently, the label in that node is assigned, together with its probability. Also, any statements before or after the label-assignment are executed in the proper order.

A conflict arises if two phases assign the same label (a label with the same name). We resolved this ambiguity by assuming that the reference is to the old label (the one assigned in a former phase) until the new label is assigned, and to the new label thereafter.

CHAPTER 6

PREDICATE EVALUATION

Three new predicates have been added to C and in this chapter we will describe how they are evaluated. There is a definite distinction between the evaluation of the in-operator and the forall- and exist-quantifiers. The in-operator is macro-expanded within the if-statement that contains the predicate. This cannot be done for the quantifiers because their evaluation is more complex. We first cover the macro-expansion and then look at the code that is produced for the quantifiers.

The general form of the in-operator,

```
expression in [constant, constant, ..., constant]
```

is transformed into the following code:

```
(expression == constant || expression == constant || ... || expression == constant)
```

A special case occurs if one of the constants represents a range of values, that is, if we have:

```
expression in [..., lower_limit..upper_limit, ...]
```

In that case, the corresponding transformation is

```
... || lower_limit <= expression && expression <= upper_limit || ...
```

This in-line expansion retains the property of standard C that sub-expressions are evaluated only if necessary. Evaluation stops as soon as the outcome of the expression has been determined.

Since the evaluation of the forall- and exist-quantifers involves not only expressions but also statements, they cannot be macro-expanded within an if-statement. Therefore, the conditional part of a rule is broken into its constituent parts if one of these parts is a quantifier. Breakpoints are the conjunctions and disjunctions of the expressions that make up the condition. With the exception of quantifiers, the individual expressions are enclosed unaltered in if-statements and connected with gotos and labels in such a way that the lazy evaluation property of C's &&- and ||-operators is preserved. For example, the disjunction of a and b,

```
if (a  ||  b) ...;
```

would result in the following code:

```
if (a) goto succ; else goto or;
or:
if (b) goto succ; else goto fail;
succ:
...;
fail:
```

Similar code is produced for the conjunction of two expressions.

In the case of quantifier-expressions, the if-statement tests a global boolean variable. The if-statement is preceded with a few lines of code that evaluate the quantifier-expression and store the result in the variable. For example, the conditional part of rule "two_2" of the system in Chapter 4,

```
?->group == MAMMAL  &&
there_exists property in ?->properties : property == HAPPY
```

will result in the following code:

```
if (?->group == MAMMAL) goto and; else goto fail;
and:
for (boolean = 0, i = 0; i < len(?->properties); i++)
  { property = ?->properties[i];
    if (property == HAPPY)
      { boolean = 1;
        break;
      }
  }
if (boolean) goto succ; else goto fail;
```

Similar code is produced for the forall-quantifier. Again, it is not hard to verify that this form of evaluation preserves the lazy evaluation property of the logical conjunction and disjunction in standard C.

Normally, references to labels that were assigned earlier can only be made in the context of the current record. However, if a quantifier's conditional expression tests a record other than the current record, then the tests try all nodes from the record's label tree. If any label in this tree satisfies the condition, the expression is considered to be true. For an example, see the last rule in Chapter 11.

CHAPTER 7

THE LIST DATA TYPE

In LISP, lists are the fundamental data type. Both data and programs are represented as lists and all computing is viewed as list manipulation. In our system, however, we provide lists simply as a convenient means of storing and processing data sequences of variable length.

The main advantage lists have over comparable data types, such as arrays, is their variable length. Lists, therefore, can never be too short or too long. But this strength turns into a handicap when lists must be implemented. Because the length of a list can vary at runtime, a compiler cannot predetermine the storage required for the list, but must provide a runtime storage allocation scheme. Furthermore, list operations are computationally expensive. Depending on the implementation, some operations are O(1), but usually more are O(N). Myers [14] has shown that it is possible to implement applicative list operations with a worst-case performance of O(lgN). We used his approach to provide the user with lists in our system.

Lists in the new language can be of any type. They can contain simple data types such as integers, floats and characters, but they can also hold pointers, arrays, structures, unions, functions and other lists. Since it is not possible in C to determine the actual size of a data structure at runtime,

these elements themselves cannot be copied into the list. Therefore, lists maintain only pointers to the data, but do not contain the data itself. One consequence of this specific implementation is that lists share their data with the original variable. Any change made to that original will affect the contents of the list as well. Furthermore, lists cannot contain numeric or character constants, since in C the address of those constants cannot be obtained.

Lists are implemented as pointers to a list data structure. Before a variable of type list can be used, this data structure must be initialized. In general, the preprocessor takes care of the initialization for the user. If, however, structures containing lists are allocated dynamically, the preprocessor cannot detect this and therefore the user is responsible for the proper initialization. This is done by assigning the expression "INC(NIL)" to the variable.

When a list is no longer needed or cannot be accessed any more (e.g. on block exit), its data structure must explicitly be freed in order to conserve space. The preprocessor will do this automatically for most cases, but again the user is responsible for dynamically allocated lists. To free a list, the function DEC() must be called with the variable in question as its argument.

Between initialization and destruction, lists can be accessed and manipulated in a variety of ways. List constants can be created, lists can be assigned to or concatenated with other lists. Sublists of lists can be accessed

as well as individual elements of a list and one can ask for the current length of a list. All these operations are implemented as function calls, where the individual functions follow the approach in Myers [14].

When using lists, their implementation with pointers should be kept in mind. The user should be aware of the fact that a change in one element of a list will result in the same change for the original (and possibly for any other list that contains the element) and vice versa. In addition, if an element becomes inaccessible (e.g. due to block exit), the list does not contain valid data any more, even if it is globally declared itself.

Since the implementation requires the initialization of list variables by the preprocessor, the user cannot initialize such variables together with their declaration. Furthermore, static lists are not supported. Globally declared lists must be used instead.

CHAPTER 8

INPUT / OUTPUT

To free the user from the tedious task of having to write their own input- and output-functions for every new rule-based system, a standard interface has been provided.

The syntax for the input is derived from the initialization syntax for external variables. The input consists of a list of records, enclosed in curly braces and separated by commas. The records themselves are enclosed in curly braces and contain the fields in the order they were specified in the input declaration. The fields are again separated by commas. If they contain simple variables, the fields are represented by their numeric or character values, respectively. Arrays and lists have the same syntax: their elements are enclosed in curly braces and separated by commas. An exception are arrays of characters; they are input in the conventional manner as a quoted string. If the records contain fields that are structures themselves, the same syntax applies to those fields as to the fields of the original records.

For the implementation of this syntax, the I/O-package of standard C, "<stdio.h>" was used, in particular the scanf function. The input function is created using the type information maintained by the preprocessor. For each record, the input function first allocates the necessary space

and initializes any list variables. It then scans the standard input for the syntax characters and the input data, which are read into the allocated space. The record is then added to the list "INPUT" and the next record is read, until the input is exhausted.

As an example, we present the slightly simplified code for the input-function from the rule-based system from Chapter 4.

```
input()
{ INPUT = [];
  scanf(" { ");
  while (next character != '}')
    { if (not first time through loop) scanf(" , ");
      ? = pointer to space allocated for current record;
      scanf(" { ");
      scanf("%d",&?->id);
      scanf(" , ");
      ?->properties = [];
      scanf(" { ");
      while (next character != '}')
        { if (not first time through loop) scanf(" , ");
          i = pointer to space allocated for one integer;
          scanf("%d",&i);
          ?->properties *= i;
        }
      scanf(" } ");
      INPUT *= ?;
    }
}
```

The output function is created in a similar manner, this time using the printf function of the I/O-package. For each record, it echoes the input data and then prints the label tree that resulted from the classification process. In order to improve readability, the curly braces around the INPUT-

list and the individual records are omitted in the output. In addition, each field of the input record is printed on a separate line. Together with each record, the labels are printed with their probabilities and rules that generated them. The printout of the rules is indented in such a way that the structure of the label tree can be clearly seen.

Users familiar with both C and the implementation of the new system have the option of writing their own, customized output function. If a function declaration for "output" is submitted, the system does not generate the standard function.

Again, we will give the (simplified) code for the rule-based system from Chapter 4 as an example.

```
output()
{ int i, j;

 for (i = 0; i < len(INPUT); i++)
  { ? = INPUT[i];
    printf("RECORD #%d:\n",i+1);
    printf("INPUT:\n");
    printf("%d\n",?->id);
    printf("{ ");
    for (j = 0; j < len(?->properties); j++)
      { printf("%d",?->properties[j]);
        if (j != len(?->properties)-1) printf(", ");
      }
    printf(" }\n");
```

```
    printf("OUTPUT:\n");
    for (all labels (group) assigned in the first phase)
      { printf("   LABEL: %d",group);
        printf("   PROB: %lf",prob(group));
        printf("   RULE: %s\n",rule_id);
        for (all labels (animal) assigned in the second phase)
          { printf("      LABEL: %s",animal);
            printf("   PROB: %lf",prob(animal));
            printf("   RULE: %s\n",rule_id);
          }
      }
  }
}
```

CHAPTER 9

PROBLEM SPECIFICATION

Automatic recognition and labeling of features in remotely sensed images is not a trivial problem. There are a variety of reasons for this:

- Features in digital images are not clearly defined, but must be determined from often noisy or ambiguous data.
- The same class of features may have different characteristics even in the same image, due to factors such as natural variability, season, weather or solar angle.
- In manual classification, the information used by the interpreter comes from many different sources. Experience, previous knowledge of the terrain covered, and complex deductions about relationships between features are just a few examples. It is hard for any one classification program to incorporate all of this "external" information.

The traditional approach has been to try solving the problem using image processing techniques. Since algorithms for techniques such as segmentation and multispectral classification do not contain any information about the context of the image, their use for interpretation purposes is limited. Despite this drawback, image processing techniques still remain necessary low-level tools for preparing digital images for knowledge-based

interpretation. This aspect of preprocessing images will be covered in the next chapter.

With the advent of rule-based and/or expert systems, it was natural that attempts have been made to employ these promising techniques to interpret images. McKeown, Harvey, and McDermott [13] describe such a system for the interpretation of airport scenes. Ohta [15] presents a system for knowledge-based interpretation of outdoor scenes and Goldberg, Goodenough, Alvo, and Karam [6] give an example of a hierarchical system for updating forestry maps. Application of all of these systems is limited, because they have to severely restrict the types of scenes that can be interpreted. Most recently, Perkins, Laffey and Nguyen [17] discuss a system for rule-based interpretation of aerial photographs using the Lockheed Expert System.

The reasons for the limitations mainly lie in the vast amount of knowledge that must be stored, updated and applied to the image. In order to correctly identify a feature in a satellite or aerial image, the knowledge base must contain information about the shape and the size of the object (both absolute and relative), its spectral characteristics (which may vary with the time of the year and/or the solar angle), its location relative to other objects, and much more. If we now consider all the features that could possibly occur in such an image, it is not surprising that restrictions are in order.

For the purposes of this thesis, we consider it sufficient to show that our system is suitable for the application — we do not attempt to do a complete classification of a digital image in every detail. Therefore, we will take selected windows out of a digital scene, preprocess them as described in the next chapter, and then try to automatically produce a reasonable and consistent interpretation of the main features within those windows.

CHAPTER 10

IMAGE PREPROCESSING

Unless they have been created under optimal conditions in a laboratory, digital images, especially aerial and satellite images, have some undesirable characteristics that make them unsuitable for immediate use in interpretation systems.

According to Schowengerdt [22], we can classify these image properties into two categories:

1. Systematic and random errors that occur during the acquisition of the image, such as noise in the scanning and/or transmission devices and geometric distortion due to the scanner or platform altitude.
2. Intrinsic characteristics that hinder the classification process. Examples are insufficient spatial resolution, or contrast that is too low in some parts of the image and too high in others.

For satellite images from Landsat, for example, problems in the first category are dealt with by the agency that distributes the images. Therefore, we will not concern ourselves with such errors.

The second category, however, is a more serious problem. There are image processing techniques available that can reduce the undesirable characteristics and visually enhance the image. These techniques for image

enhancement, however, are useful only for manual classification. Since they do not create any new information, but only change the "look" of the image, they do not help in automatic interpretation.

For a good introduction to image processing techniques see Schowengerdt [22]; for more detailed descriptions of algorithms see Rosenfeld and Kak [19], or Pratt [18].

For our example, we used a digitized aerial color infrared image of Tucson, Arizona (Fig. 5). This is a three-band image, 512*512 pixels, with a ground resolution of approximately 4 m. The image was processed on a VAX-11/750 minicomputer. The software used for the image processing consists of a large package of FORTRAN subroutines, the "System at Arizona for Digital Image Experimentation" (SADIE, [21]) and International Imaging System's interactive image processing package (I2S, [8]).

First, the image was submitted to an unsupervised statistical multispectral classifier (an implementation of the nearest-mean algorithm), that created a new image with eight different classes of pixels. In addition, the classifier also provided the global statistics (mean and variance of the pixels in the three frequency bands) for each of the classes. Pixel noise in the classified image was reduced with a local majority filter (Schowengerdt, [22], pp. 188-190). The resulting smoothed, spectral cluster map constituted a segmented image. We extracted the data for two windows of 128*128 pixels from the image shown in Figure 5. Figures 6 and 7 show the smoothed spectral cluster maps for the two windows, with a different color assigned to

each of the eight classes. Spatially connected segments were consecutively numbered with a "blob-coloring" algorithm (Ballard and Brown, [1], p. 150). The boundaries of the segments were obtained with a modified "turtle"-algorithm (Papert, [16]) and converted from raster- into vector-format. The windows contained approximately 1700 and 3200 segments, respectively.

The input to the rule-based system is a list of segment descriptions. Each of those segments consists of a unique ID (the segment number), the mean gray levels in all three bands for the pixels in that segment, a list of integer (pixel) coordinates that describe the boundary of the segment and finally a list of the IDs of all neighboring segments.

Fig. 5: Digitized aerial color infrared photograph of Tucson, AZ.

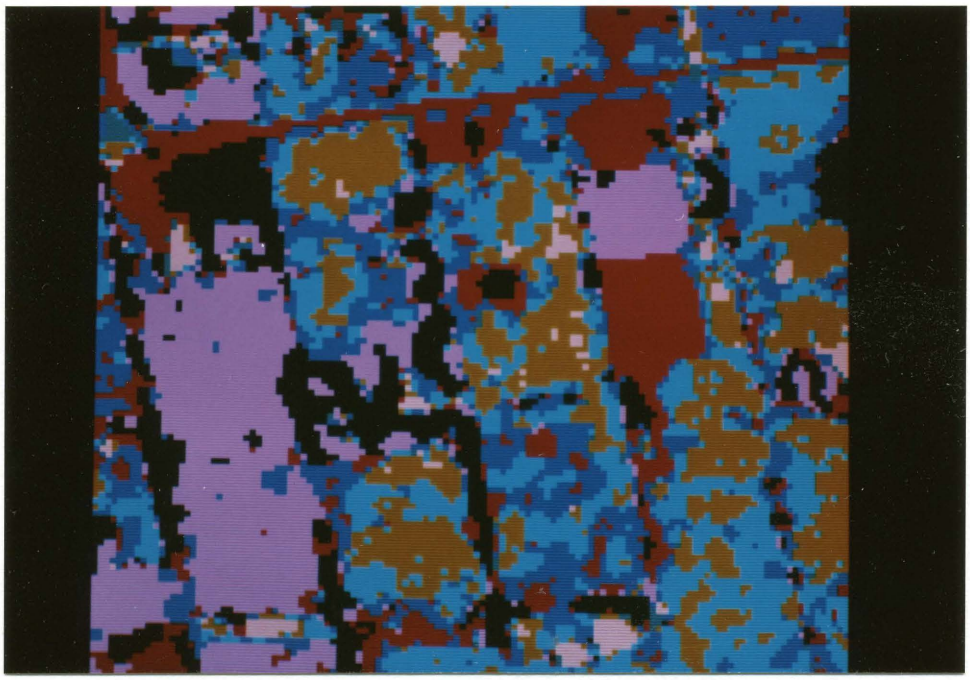

Fig. 6: Smoothed spectral cluster map of window #1.

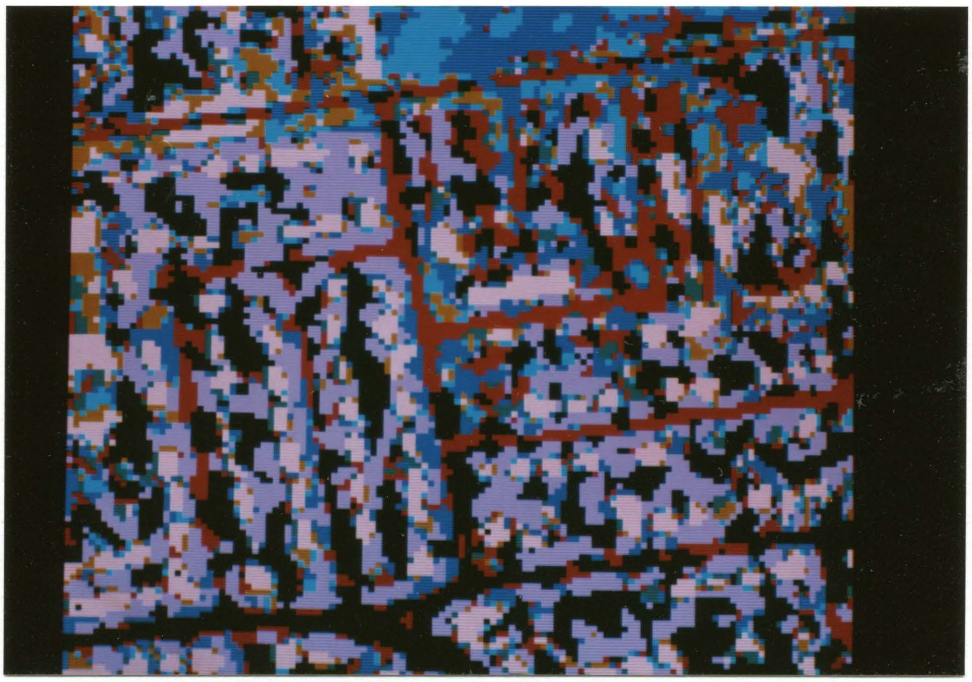

Fig. 7: Smoothed spectral cluster map of window #2.

CHAPTER 11

IMAGE INTERPRETATION

In this chapter, we cover the implementation of a rule-based system for the interpretation of aerial images. The verbal description of the input data from the last chapter translates into the following specification for an individual segment:

```
#define NBNDS    3

typedef struct {
  int x, y;                        /* pixel coordinates */
} point;

input {
  int id;                          /* unique segment ID */
  double spmean[NBNDS];            /* spectral means */
  point @boundary;                 /* pixel boundary */
  int @neighbors;                  /* IDs of neighboring segments */
} segment;
```

The classification of segments is done in three phases. The first phase, called "spect," consists of eleven rules and attempts to classify the segments into basic categories according to their spectral characteristics. The six categories used in this phase are "vegetation," "water," "soil," "asphalt," "sand," and "gravel." The following is an example of a rule for this phase:

```
rule spect_3
dist[0] <= MAXDIST[0]  =>  assign (mindist/dist[0]) SOIL;
```

The three spectral bands define a 3-D space. The rule compares the spatial distance ("dist[0]," precomputed) between the segment and one of the eight clusters ("MAXDIST[0]") determined during preprocessing of the image. If this distance is below some threshold, the label is assigned with a probability which depends on some other threshold ("mindist"). Thresholds are presently determined manually from the global spectral clustering statistics, since the gray levels in the spectral bands have not been normalized and the thresholds therefore don't apply to a different image.

The second phase, "geom," uses the geometric characteristics of the segments (derived from the description of their boundaries) to refine the classification of the first phase. The labels here are called "active agriculture," "urban vegetation," "natural vegetation," "river," "lake," "fallow agriculture," "unpaved road," "natural area," "paved road," "parking lot," "gravel road," "commercial building," and "residential building." In this phase, we use rules like the following:

```
rule geom_4
?->spl == WATER  &&  curvilinear < 0.3  &&  aspect_ratio > 10.0  =>
  assign (prob(?->spl)) RIVER;
```

Based on the classification result from the first phase ("?->spl") and on the values of the (precomputed) geometric properties "curvilinear" and "aspect_ratio", this rule assigns the label "river." The value of the variable

"curvilinear" is computed as the ratio between area and perimeterlength of a segment. It serves as an indicator for the "compactness" of the segment. The aspect ratio of a segment is the ratio of length squared to area, where length is the longest distance between any two points of the segment. This value is a good indicator for the shape of a segment: the greater the value, the longer is the corresponding segment.

The first and second phase require some numeric computation, such as the average gray levels for all pixels in a segment, or the geometric characteristics of the segments. Due to the large amount of data, these computations may become quite expensive (usually in terms of CPU-time) and it would be convenient if we could separate them from the classification. In our system, the computations for the first phase were done by the image processing system, SADIE [21]. For the second phase, no programs exist yet in that system. These computations were therefore incorporated into the rule-based system. However, with the interface provided, there would be no problem in shifting computations from one system to another.

The third and potentially most powerful phase (called "context") reassesses the probabilities assigned to the labels in the second phase. This is done by looking at the neighboring segments and increasing/decreasing the confidence values of the labels assigned in the second phase according to the overall consistency. Again, we will show an example of a rule in this phase:

```
rule context_14
?->label == PARKING_LOT  &&  for_all element in ?->neighbors :
element->label in [PAVED_ROAD,GRAVEL_ROAD,COMMERCIAL_BUILDING]  =>
  assign (min(prob(?->label)*(1+0.05*len(?->neighbors)),1)) ?->label;
```

This rule represents knowledge about spatial relationships between adjacent features in a scene. If the current segment has been classified as a parking lot, and all its neighboring segments have labels of objects that could be expected to be adjacent to a parking lot, the confidence value of the current segment is increased.

In general, certain types of objects are usually found next to some other types. If our classification supports a rule like this, the probability for the particular label is increased. We also have rules to the contrary. If the classification of an object is not consistent with the labels of its neighbors, the probability of that classification is decreased. This technique of improving the overall consistency of the classification is called "constraint relaxation" (Levine, [12], pp. 404-420). For descriptions of other approaches to the use of context in image classification see Wharton [27], or Tilton, Vardeman, and Swain [25].

In the next chapter, we present the results obtained from the data described in Chapter 10 and the rule-based system from this chapter.

CHAPTER 12

RESULTS

The preprocessor and the experimental rule-based system have been implemented on a VAX 11/785 under the UNIX operating system. The lexical analyzer and the parser for the new language are lex- and yacc-based, respectively, (Lesk and Schmidt, [11]; Johnson, [9]), and consist of approximately 260 lines of code. The preprocessor itself is written in standard C and contains about 2700 lines of code. The rule-based system was specified with approximately 300 lines of extended C. It was run through the preprocessor and expanded into a standard C program of almost 900 lines. Of those, the three phases together account for approximately 600 lines; 100 lines are used for diverse (user-defined) functions; the I/O is responsible for the remaining code. If the rule-based system was hand-coded directly in standard C, the resulting program would be shorter by a factor of one third, mostly due to savings in the I/O portions of the code and in the evaluation of predicates. The C program generated by the preprocessor was compiled under the UNIX C-compiler with full optimization. The timings in Table 1 show that the lexical analyzer, parser and preprocessor run quickly. The whole process takes under one minute of real time, and the C-compiler alone accounts for approximately 80% of that amount.

Table 1: Preprocessor Timing

	real time	user time
Lexer	3.2 sec	0.2 sec
Parser	2.7 sec	1.4 sec
Preprocessor	4.6 sec	1.2 sec
C-compiler	36.3 sec	28.0 sec

Next, the resulting C program was linked with the code that implements the diverse list functions (approximately 230 lines), and run on the test data described in Chapter 10. During the first and second phase, between two and three labels per segment and phase were assigned on the average. The third phase only reassesses the probabilities of the labels in the second phase, so it does not continue this expansion. The first window with 1662 individual segments was processed in 3.0 minutes of real time, and the second window (3224 segments) took 5.2 minutes. Again, these times are quite reasonable, especially if compared to some of the image processing routines (the "turtle"-algorithm, for example, takes about 10 and 16 minutes of real time, respectively). To show the superiority of our system over an interpreted system, the rules were implemented in LISP and run on the same test data. Our system proved to be faster by a factor of about 260! Due to problems with the LISP compiler, no comparison could be made between C and compiled LISP.

To determine the efficiency of the system, a profile of the running program was obtained, using UNIX profiling software. The results are summarized in Table 2.

Table 2: Profile of executing system

	cpu time
List functions	43.5 %
Output	30.0 %
Input	10.5 %
Phase III	4.0 %
Phase II	3.5 %
Phase I	2.5 %

Almost one half of the total time is used by the different list functions, which reflects their heavy use. The same thing can be said of the time that is spent for the output; this amount could be reduced by writing a customized, less verbose output function.

In terms of the classification results, things can still be improved. The first phase gives the best results; the classification is quite good for the second window and acceptable for the first window, where the spectral classes are harder to distinguish. In the second phase, results are not yet satisfactory. For this, there are two distinct reasons. For one, the segmentation process does not deliver the optimum separation of the segments. Segments that represent different features in the image are merged together and others are split up unnecessarily. The other reason is the computation of the geometric properties of the segments. Here, some experience is needed until the thresholds can be set up properly. The third phase looks at the neighboring segments to confirm or reject existing classifications. However, the forall- and exist-quantifiers, which are heavily used in this phase,

might not be quite appropriate for "real world" problems. What may be needed here are "most"- and "few"-quantifiers, and this could be achieved by having "forall" and "exist" return probabilities instead of binary values.

The results of the classification process are shown in Figures 8 and 9.

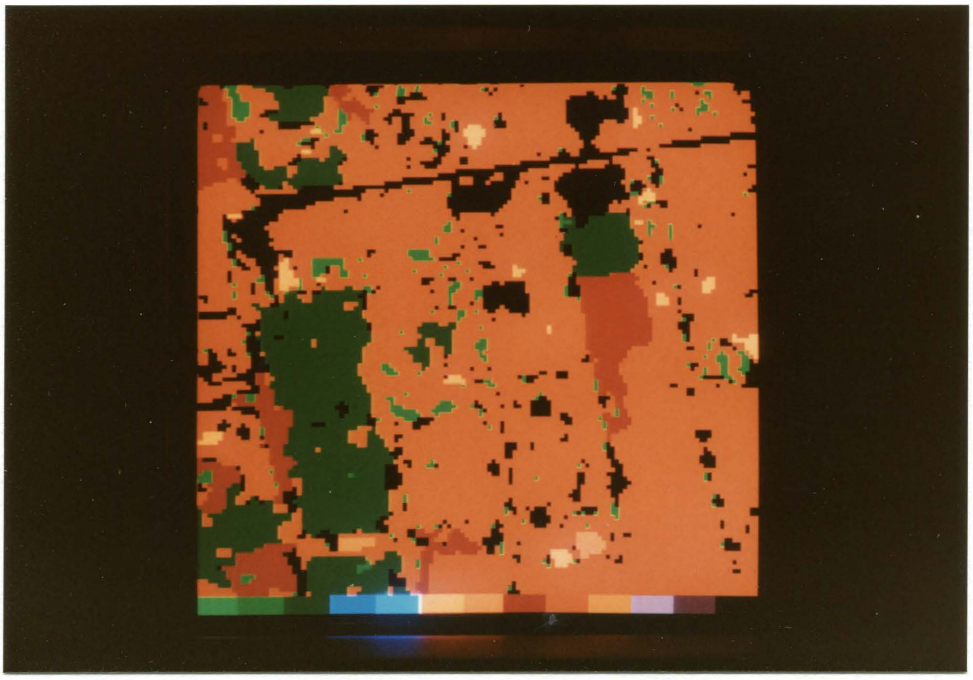

Fig. 8: Classification map for window #1

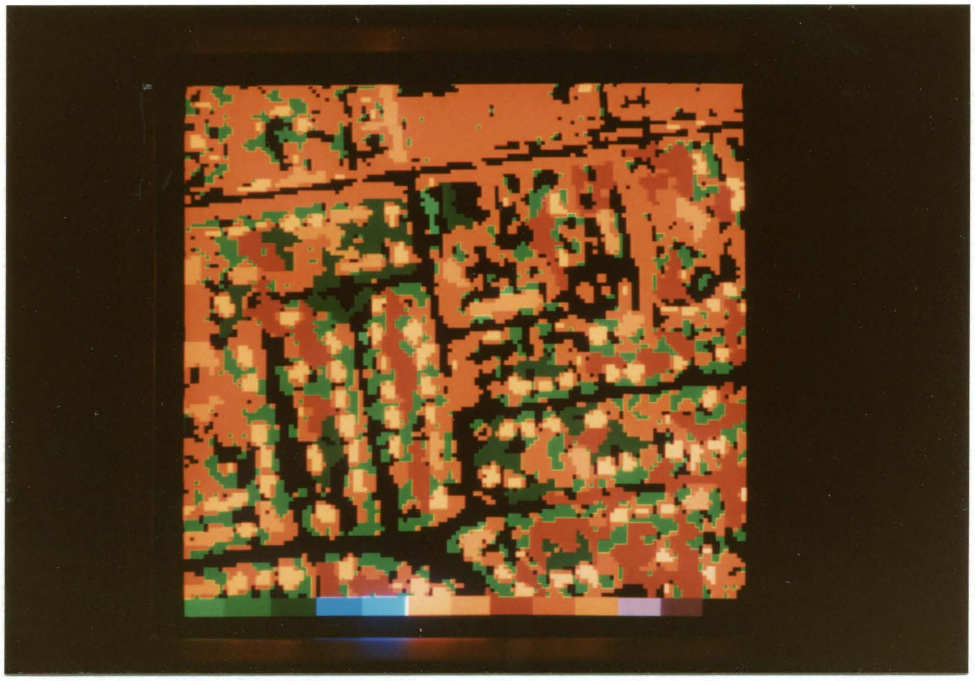

Fig. 9: Classification map for window #2

CHAPTER 13

CONCLUSION

A system for the automatic generation of rule-based systems has been presented. This system has some advantages over similar systems that have been developed so far. Mainly, the new system is based on C, a language that is widely used in technical and production environments and with which many programmers are familiar. The implementation of this system has proved to be feasible and more efficient than comparable interpreted systems. But as with every experimental system, there are areas that could be improved.

In most cases, the implementation was straightforward. Therefore, in terms of efficiency, a lot may be gained if more sophisticated approaches are taken. An example are function calls in the conditional parts of rules. They are evaluated every time control reaches the rule. But often these functions compute only properties of the current record that do not change with different paths through the label tree (as in phase II of the application). The system would be more efficient if such loop-invariant portions of the code could be moved outside the loop over the label tree.

Efficiency could also be gained if the predicates were not evaluated one by one as they are encountered, but if instead common subexpressions

were filtered out and evaluated only once. Furthermore, the results of expressions that are invariant over multiple iterations could be saved, thus further reducing the amount of computation.

The syntax of the input could be simplified and generalized. For lists of characters, the same "quoted string" syntax should be employed as for character arrays. It might be desirable to be able to input not only structures but unions as well. Furthermore, users might wish to input their data in arbitrary order or without the annoying brackets.

The diagnostics issued by the parser, lexical analyzer, and preprocessor also need some improvement. Currently, the parser and the lexical analyzer produce one generic error message, and the preprocessor outputs only a few more diagnostics.

But despite those deficiencies, the new system has proven to be quite useful in the application and further refinements appear likely.

APPENDIX

REFERENCES

[1] BALLARD, D. and BROWN, C.
"Computer Vision."
Prentice-Hall, 1982

[2] BUCHANAN, B., SUTHERLAND, G., and FEIGENBAUM, E.
"Heuristic DENDRAL: A program for generating
explanatory hypotheses in organic chemistry."
Machine Intelligence, Edinburgh University Press,
Vol. 4, 1969, pp. 209 - 254

[3] CLOCKSIN, W., and MELLISH, C.
"Programming in Prolog."
Springer Verlag, 1984

[4] CROSS, G.
"Tools for constructing knowledge-based systems."
Optical engineering, Vol. 25, No. 3, March 1986, pp. 436 - 444

[5] DUDA, R., HART, P., NILSSON, N., and SUTHERLAND, G.
"Semantic network representations in rule-based inference systems."
In: WATERMAN, D., and HAYES-ROTH, F. (eds.)
Pattern-directed inference systems, pp. 203 - 221
Academic Press, 1978

[6] GOLDBERG, M., GOODENOUGH, D., ALVO, M., and KARAM, G.
"A Hierarchical Expert System for Updating Forestry Maps
with Landsat Data."
Proceedings of the IEEE, Vol. 73, No. 6, June 1985, pp. 1054 - 1063

[7] HAYES-ROTH, F., WATERFORD, D., and LENAT, D.
"Building Expert Systems."
Addison-Wesley, 1983

[8] I2S
"System S570 Digital Image Processing System,
Version 3.0 User's Manual."
International Imaging Systems, 1984

[9] JOHNSON, S.
"YACC - Yet Another Compiler Compiler."
in: UNIX Programmer's Manual
University of California, Department of Computer Science, 1984

[10] KERNIGHAN, B., and RITCHIE, D.
"The C Programming Language."
Prentice-Hall, 1978

[11] LESK, M., and SCHMIDT, E.
"LEX - Lexical Analyzer Generator."
in: UNIX Programmer's Manual
University of California, Department of Computer Science, 1984

[12] LEVINE, M.
"Vision in Man and Machine."
McGraw-Hill, 1985

[13] McKEOWN, D., HARVEY, A., and McDERMOTT, J.
"Rule-Based Interpretation of Aerial Imagery."
IEEE Transactions on Pattern Analysis and Machine Intelligence,
Vol. PAMI-7, No. 5, September 1985, pp. 570 - 585

[14] MYERS, E.
"AVL DAGS."
University of Arizona, Department of Computer Science,
Technical Report No. 82-9

[15] OHTA, Y.
"Knowledge-based Interpretation of Outdoor Natural Scenes."
Pitman Advanced Publishing Program, 1985

[16] PAPERT, S.
"Uses of technology to enhance education."
Massachusetts Institute of Technology, AI Lab, Technical Report No. 298

[17] PERKINS, W., LAFFEY, T., and NGUYEN, T.
"Rule-based interpreting of aerial photographs
using the Lockheed Expert System."
Optical Engineering, Vol. 25, No. 3, March 1986, pp. 356 - 362

[18] PRATT, W.
"Digital Image Processing."
John Wiley and Sons, 1978

[19] ROSENFELD, A., and KAK, A.
"Digital Picture Processing."
Academic Press, 1976

[20] RUSSELL, S.
"The Compleat Guide to MRS."
Stanford University, Department of Computer Science,
Report No. KSL-85-12

[21] SADIE 3.1
"User's Manual."
"Subroutine Reference Manual."
University of Arizona, Digital Image Analysis Laboratory, 1985

[22] SCHOWENGERDT, R.
"Techniques for Image Processing and Classification in Remote Sensing."
Academic Press, 1983

[23] SHORTLIFFE, E.
"Computer-based medical consultation: MYCIN."
American Elsevier, 1976

[24] STEELE, G.
"Common LISP Reference Manual."
Digital Press, 1984

[25] TILTON, J., VARDEMAN, S., and SWAIN, P.
"Estimation of Context for Statistical Classification
of Multispectral Image Data."
IEEE Transactions on Geoscience and Remote Sensing,
Vol. GE-20, No. 4, October 1982, pp. 445 - 452

[26] VAN MELLE, W.
"A domain-independent system that aids
in constructing knowledge-based consultation programs."
Stanford University, Department of Computer Science,
Ph.D. Dissertation, Report No. STAN-CS-80-820

[27] WHARTON, S.
"A Context-Based Land-Use Classification Algorithm
for High-Resolution Remotely Sensed Data."
Photographic Engineering, Vol. 8, No. 1, February 1982, pp. 46 - 50

[28] WEISS, S., and KULIKOWSKI, C.
"EXPERT: A system for developing consultation models."
Proceedings of the 6th International Joint Conference
on Artificial Intelligence, 1979, pp. 942 - 947