1980-3

Programmer's Guide to the Eleusis Program

Thomas G. Dietterich

Internal Report
May 20, 1980
Intelligent Systems Group
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

TABLE OF CONTENTS

1. Introduction

This document is a companion to the thesis and user's guide (appendix II of the thesis) which describe the Eleusis program. Programmers who seek to understand, maintain, and improve the Eleusis program should find this document helpful in elucidating the data structures, procedures, flow of control, and (alas) kludges in the program.

2. Overview

The Eleusis program is a large (9616 lines, 143 procedures) PASCAL program. Fortunately, it is broken down into 5 layers which are similar to each other in many ways. Each layer performs 3 basic functions:

a) Learning Element (LE) - which discovers plausible descriptions of the layout.

b) Performance Element (PE) - which develops a description of the set of legal cards which could extend the sequence.

c) Critic (CR) - which tests a description of the layout to see that all correctly played cards satisfy the description and that no incorrectly played card satisfies the description.

As described in the thesis, the execution of each task (LE, PE, CR) involves descending through the layers from layer 5 down to layer 1 and then returning. For example, in the LE task, each layer performs three basic steps:

1. Transform the data structures to the representation appropriate to this level.
2. Search the space of possible descriptions at this level by invoking the level immediately below to process the data.
3. Evaluate the results of the search to eliminate implausible rules.

The PE and CR tasks operate in similar ways.

The data structures used by the program are described in section 3. Section 4 describes the three tasks of the program (LE, CR, and PE) and discusses the data transformations and flow of control of each task. Section 5 describes the lexical analyzer and parser that make up the user interface to the program. And finally, section 6 provides advice on maintaining the program and suggestions for solving some of the more blatant problems of the present implementation.

This document should be read in conjunction with a source listing of the program. As noted in the program preamble, the program is divided into 9 sections. References will be made to these sections throughout this programmer's guide.

3. Data Structures and Data Transformations.

This section is best read in conjunction with section 2 of the program source. The program has 3 fundamental structures: symbol tables, layouts, and rules. The layouts are sequences of events, the rules are possible descriptions of the layouts, and the symbol tables define the meaning of the variables in both the layouts and the rules. Let us start with the simplest data type: layouts.

3.1 Layouts (`tsevents`)

Conceptually, a layout is a sequence of events laid out exactly as cards are laid out in Eleusis. The fundamental type is the `tsevent`, a packed record with the following fields:

`complex`: a canonical VL_1 complex describing the event

`negcomplex`: a linked list of canonical VL_1 complexes describing negative events which followed the positive event represented by `complex`.

`nextevent`: a link to the next `tsevent` in the layout.

`segstart`: a link to a `tsevent` in another layout (see below).

Thus, the Eleusis layout:

AC	2H	5D
	3D	
	4S	

is stored internally as a singly-linked list of `tsevent` records as in Figure 1.

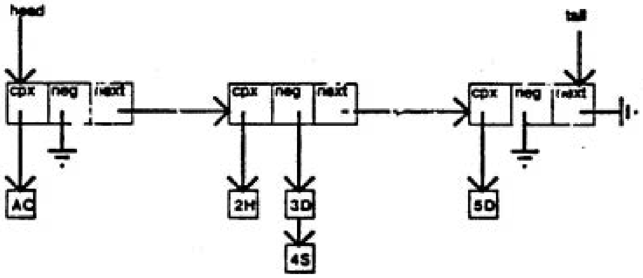


Figure 1.

Each `tsevent` is represented by 3 boxes, one each for the `complex`, `negcomplex`, and `nextevent` fields (`segstart` is omitted). Each layout has a head and tail pointer. The head and tail have the following names:

Level	Head Variable Name	Tail Variable Name
5	`layout`	`layouttail`
4	`l4layout`	`l4layouttail`
3	`l3layout`	`l3layouttail`

The head points to the first `tsevent` in the layout. The tail points to the last `tsevent`.

Exceptions

During level 2, the negative events are shifted right one `tsevent` so that each `tsevent` contains all cards which were played after the positive event in the previous `tsevent`. The procedures `shiftnegcomplex` and `unshiftnegcomplex` perform and undo this shift. When negative events are shifted, it is possible to have a `tsevent` which contains no positive complex. `Shiftnegcomplex` must create this `tsevent` and `unshiftnegcomplex` must destroy it.

During `level2pe`, the `l2layout` is temporarily distorted by appending a dummy event to the tail of the layout. In this case, the `l2layout` tail pointer is not updated to point to this dummy event. This modification is very temporary and `l2layouttail` is used to remove the dummy event when it is no longer needed.

3.1.1 VL_1 Complexes.

Layouts point to lists of VL_1 complexes. Conceptually, a VL_1 complex is a conjunction of VL_1 selectors. Each selector involves a *variable*, a set of values called the *reference*, and a *relation* between the variable and the set of values. In data structure terms, a `vl1complex` is an array of `vl1selectors` and each selector is a record which contains fields for the relation and the reference of the selector. The variable corresponding to the selector is implicit in the position of the selector in the array. A `vl1complex` can only be interpreted if it is associated with a `vl1symboltable`. The symbol table defines the characteristics of each VL_1 variable. The ith VL_1 variable in the given symbol table has values specified by te ith selector in the `vl1complex`.

The precise layout of a `vl1selector` is:

`reference`: a PASCAL set of values from 0 to `maxvalue`. Each VL_1 variable has a specific set of values, its domain. For example, the *suit* of a card has the domain {*clubs, diamonds, hearts, spades*} . The set elements 0 through 3 are used to represent these reference values. The value *, denoting *any* value, is precisely the set [0..3]. Thus, the representation of * differs from variable to variable.

`relation`: a scalar type specifying the relationship between the value and the reference. `Relation` has an intimate connection with the `printref` fields. Usually, this relation field has the value `releq` (equals).

`useprintref`: a boolean. If this is `true`, then the values specified in `printref` must be used to interpret the selector. However, the `reference` field is always set properly too.

`printref`: an array of two values used as described below.

For example, the selector [$value > 10$] is represented by the selector record:

`reference=` [10, 11, 12] (this is not an error, see biassing below)
`relation=` `relgt`
`useprintref=` `true`
`printref[1]=` `10`
`printref[2]=` don't care

The selector [$suit$ = *clubs..diamonds*] is represented as:

`reference=` [0, 1]
`relation=` `releq`
`useprintref=` `true`
`printref[1]=` `0`
`printref[2]=` `1`

In general, if `useprintref` is `true`, only the first element of `printref` is used except when the `relation` is `releq` and the variable is a `linear` (or `dlinear`, `clinear`, etc.) domain.

Each VL_1 complex contains the array of selectors called `selectors`. It also contains:

`nextcomplex`: a link to the next complex when the complexes are linked in a linked list

`play`: indicating what play this card was played during. The first card played is play 1. Each card in a string of cards played in one turn is given the same value for `play`.

`fp, fq`: flags used during the A^q algorithm.

Interpretation of `vl1complexes` is tied directly to the `vl1symboltables` defined at each layer:

Layer	Symbol Table Name
5	none (see exceptions)
4	`l4symboltable`
3	`l3symboltable`
2	`l2symboltable`

Exceptions:

At layer 5, a `vl1complex` is referred to in the program (in the commentary) as a `simple vl1complex` or `svl1c`. A simple VL_1 complex contains only 2 selectors, one for `suit` and one for `value`. The variables are permanently fixed at indices `vl1valuex` and `vl1suitx` in the `vl2symboltable` at level 5 (and in the `l4symboltable` at level 4).

At level 4, `vl1complexes` are sometimes used to describe segmentation conditions. Segmentation conditions are always defined as delta variables, but unfortunately, delta variables are not defined at level 4 in the `l4symboltable` but only at level 2 in the `l2symboltable`. The solution (which has other advantages which are described below) involves using `l4symboltable` to define these delta variables such that where `l4symboltable` says that the variable is `color` (for example), we interpret it to mean `dcolor` (delta color). Thus, the domain definitions for the `l4symboltable` are incorrect when applied to segmentation conditions. This causes other problems and is the principle reason why you cannot print out VL_1 segmentation complexes.

When `vl1complexes` are used in `vl1rules`, the selectors are always interpreted using the `l2symboltable` (except for the segmentation conditions which_are interpreted using the `l4symboltable`).

VL_2 complexes are described in 3.4 in a section devoted to the peculiarities of VL_2.

3.2 Symbol Tables

The second major structure which we must consider is the symbol table. A symbol table in Eleusis defines the semantics of each variable and its relationships to other variables. Much of the information in the symbol table is crucial to the inference capabilities of the Eleusis program. There are 4 different symbol tables in the program (one each in levels 2, 3, 4, and 5). One could imagine combining these and eliminating a good deal of redundancy. However, it is important, in the knowledge-layer methodology, to isolate the layers from each other as much as possible. A common symbol table would destroy the independence of the layers.

The most basic symbol table type is the `vl1symboltable`. It is merely an array of records called `vl1variables`. One `vl1variable` is defined for each VL_1 variable in the particular problem at hand. In particular, the variables in the `vl1symboltable` are in one-to-one correspondence with the selectors in the `vl1complexes` at any given layer of the program.

A `vl1variable` is decomposed into three subparts. The first subpart is the `name`. This is a 10 character name which describes the variable. A `d` is prepended to the name if the variable is a "delta" variable (difference variable). An `s` is prepended if the variable is a "sum" variable. Subscripts are appended to the name according to the subscripting convention described in the thesis. A subscript of `0` indicates that this variable refers to the current card. A subscript of `1` refers to the card immediately preceeding, and so on.

The second subpart of a `vl1variable` is the `domain`. This is a record of information describing the domain set of the variable. It includes the fields:

`dtype`: the domain type of the variable. There are 8 domain types in Eleusis which are described below in section 3.2.1.

`max`: This gives the largest element of the domain which legally exists in the reference. In other words, the domain is represented by the set $\{0, 1, \ldots, max\}$. There are `max+1` elements in the domain.

`bias`: This gives an amount which is to be added to the reference values before they should be printed out. It is only relevant for numeric valued variables. Example: the value of card is represented by the domain `dtype=linear, max=12, bias=1`. In other words, a card can assume values from the set $\{A, 2, 3, 4, \ldots, 9, 10, J, Q, K\}$. This is represented by the PASCAL set [0..12] with a bias of 1.

`zero`: This indicates which element of the domain corresponds to zero. This is redundant information since it could be computed from `max` and `bias`. It is used for generalizing `dlinear` variables.

`cors`: This indicates whether this variable describes cards or describes strings of cards. The only variables that describe strings of cards are `length` and derived descriptors based on `length`. `Cors` basically tells how to interpret the bits in the `refsyms` table (see below).

The third subpart of the `vl1variable` is `advice`. This is a record of information which provides advice to the program about which variables to derive and about how plausible they are.

`plausibility`: a real number between 0 and 1. `Plausibility` is used to compute cost functions in `aqcost` and distances in the decomposition algorithm (procedure `mergeunions`).

`gen`: a set of `derivecode`. `Gen` tells level 2 whether or not it should generate `sum` or `difference` variables based on this variable. `Gen` also indicates if this variable is a `builtin` variable (e.g. *value, suit,* or *length*) or a derived variable. The `bitset` element of `gen` indicates that this variable can be derived using the `values` sets in `refsyms`. In the current program, `bitset` is always set (all variables have `refsyms` entries).

3.2.1 Domain Types

The Eleusis program supports eight different domain types. There are three fundamental types: nominal, linear, and clinear. The remaining types are based on sums and differences of these fundamental types.

`nominal`: Values in a `nominal` domain have no particular relationship to each other. A `nominal` variable can be generalized by extending the `reference` by adding any value from the domain.

`linear`: Values in a `linear` domain set are totally ordered. They are also considered to be equally spaced. The reference of a `linear` variable can be generalized by closing intervals and creating one-sided intervals. For example, $[value = 3, 6]$ may be generalized to $[value = 3, 4, 5, 6]$. $[value = 8, 13]$ may be generalized to $[value > 7]$ (since 13 is the largest value in the domain).

`clinear`: Values in a `clinear` domain set are cyclically ordered. The reference of a `clinear` variable can be generalized by closing the shortest interval in the set. For example, $[suit = clubs, spades]$ can be generalized to $[suit = spades..clubs]$ (an end-around interval).

Difference variables represent the change between two events. Special domain types are defined for "difference" or "delta" variables.

`dnominal`: A `dnominal` variable has the domain $\{0, 1\}$. It takes value 0 if the two events have the same value of this variable, otherwise a `dnominal` variable takes the value 1. There are no simple generalizations of a `dnominal` variable.

`dlinear`: A `dlinear` variable has the domain $\{-(max+1), \ldots, 0, \ldots, max+1\}$ where *max* is the `max` field of the `domain`. There are several ways to generalize a `dlinear` variable as indicated in the examples below:

$$\begin{array}{rcl}
[dvalue = -3, -1] & |\!< & [dvalue < 0] \\
[dvalue = -3, +7] & |\!< & [dvalue <> 0] \\
[dvalue = -3, +3] & |\!< & [dvalue = -3, +3] \\
[dvalue = -3, 0, +3] & |\!< & [dvalue = -3.. + 3] \\
[dvalue = -3, 0] & |\!< & [dvalue <= 0]
\end{array}$$

`dclinear:` A `dclinear` variable has exactly the same domain as the `clinear` variable from which it was computed. It is generalized in the same way.

Sum variables can be created for `linear` and `clinear` variables. The sum of `nominal` values is not defined.

`slinear:` A `slinear` variable as the value of the sum of two `linear` variables. Its domain runs from $\{2bias, \ldots, (2max) + (2bias)\}$. It is generalized in the same way as a `linear` variable.

`sclinear:` An `sclinear` variable is just the sum of two `clinear` variables (modulo $(max + 1)$). It has the same domain and the same generalization rules as the corresponding `clinear` variable.

3.2.2 Reference Symbols (Descriptor Semantics)

The `rsymbols` field in the domain definition is very important. It indexes into a table of so-called *reference symbols*. This table was originally intended to store the definitions of symbolic domain values such as "ACE" or "CLUBS." However, as the program was developed, this table become the portion of the program which defined the semantics of each variable and its relationships to other variables. These semantics are based on characteristics of the deck of cards and of typical card descriptors. Thus, they are quite domain specific. As noted in the thesis, this domain dependence causes level 3 to be less general than it should be.

All defined descriptors have a non-zero `rsymbols` field (except for dummy variables in VL_2). This points into the `refsyms` array to the start of a contiguous block of `refsyms` entries, one for each value in the domain of the descriptor.

Each entry in `refsyms` defines the set of cards in the deck which have that particular value. The type `cardset` is a set of integers which represent the cards in the deck in the standard Bridge order (AC, 2C, 3C, ..., KC, AD, 2D, ..., KD, AH, ..., KH, AS, ..., KS). Each `refsyms` entry contains the fields:

`values:` a `cardset` of the cards which have this value. For example, if the descriptor involved is *color*, and the domain value was *red*, then `values= [AD, 2D, ..., KH]`.

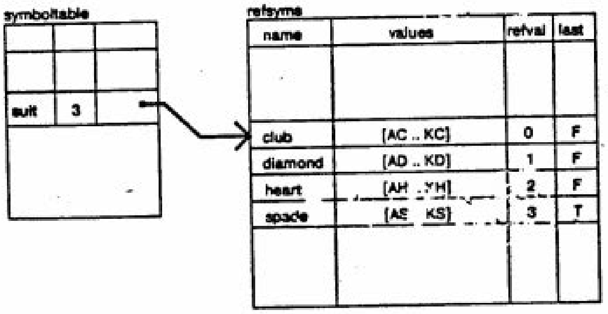
`refvalue`: The number of the corresponding domain element. This value has been biassed so that the smallest element in the domain has a `refvalue` of zero and the largest element has a `refvalue` of `domain.max`.

`last`: a boolean flag which marks the last `refsyms` entry for this descriptor. It is false for all other `refsyms` entries.

`isname`: a boolean flag indicating whether this domain value is an alphanumeric character string or an integer. If `isname` is true then the `name` field contains a character string representing the name. If `isname` is false, the `number` field contains an integer to be printed for this value.

Three basic descriptors, *suit, value,* and *length* are predefined by the procedure `initsymbols`. This is a good place to look in order to understand how the `refsyms` array is used.

When additional descriptors are defined using the `DEFINE` command, `refsyms` entries are built for their values by the procedure `v12ctoset` which converts a `v12complex` to a `cardset`.



The `refsyms` entries are used in layer 5 to define new descriptors, in layer 4 to add derived descriptors to the layout, and in layer 3 to determine which variables remain constant under the segmentation condition. Layer 3 also uses `refsyms` to derive *length*-related variables.

Exceptions

When `cors = forstring`, the interpretation of each `refsyms` entry is slightly different. Instead of interpreting `values` as a set of cards, the elements of `values` correspond to possible lengths of a string of cards. For example, if we define the descriptor `lengthmod2` to have a domain of 0 (if `length` is even) and 1 (if `length` is odd), then `lengthmod2` has two `refsyms` entries. The 0 entry has `values =`

`[0,2,4,6,...,50]` and the 1 entry has `values = [1,3,5,7,...,51]`. Layer 3 is the only layer that makes use of this interpretation.

3.2.3 Relevant Program Variables.

There is only one `refsyms` array. All four symbol tables point into this array. The four symbol tables are:

name	layer	number of used elements
`vl2symboltable`	5	`nvl2vars`
`l4symboltable`	4	`nl4vars`
`l3symboltable`	3	`nl3vars`
`l2symboltable`	2	`nl2vars`

There are many additional variables which play some role in the symbol table. Firstly, there are global variables which indicate the number of elements in each symbol table that are actually being used: `nvl2vars`, `nl4vars`, `nl3vars`, and `nl2vars`. Secondly, there are cross-reference arrays which tell how a variable in one symbol table is related to a variable in another symbol table. For example,

- `from34` is an array which tells, for a given level 3 variable, the index of the corresponding level 4 variable in `l4symboltable`.
- `to32` is an array which tells, for a given level 3 variable, the index of the corresponding level 2 variable in `l2symboltable`.

These mappings are not 1-1 or onto. For example, several level 2 variables may be defined (with different subscripts, etc.) from a single level 3 variable.

3.3 VL_1 Rules

A VL_1 rule is conceptually a disjunction of VL_1 complexes. The `vl1rule` data structure is a record which contains, among other things, a linked list of `vl1complexes` representing this disjunction. The fields of the `vl1rule` record are:

`complexes`: a linked list of `vl1complexes` interpreted using the `l2symboltable`. For the DNF and DECOMP models, this represents a disjunction of complexes. For the `PERIODIC` model, each complex is understood as describing one phase of the period. The first complex in the list describes the phase which includes the first card on the layout. The second complex describes the second phase, and so on. The fact that VL_1 rules are interpreted using the `l2symboltable` causes some difficulties. In particular, a new `l2symboltable` is built for each segmentation condition at level 3. The rules developed using one segmentation condition must be "harvested" and processed by the upper layers before the `l2symboltable` is destroyed. The solution involves the use of a coroutine linkage (a kludge) between layers 3 and 4. Just after each call to `level21e`, level 3 calls `level4examine` which inspects the `vl1rulebase` (see below) to snatch the good `vl1rules` and convert them to VL_2 before the `l2symboltable` changes. This kludge could be avoided if symbol information were stored with the rule rather than being implicit in the representation of the `vl1complex`. The tradeoff is between storage and elegance.

`segcomplex`: This is a `vl1complex` which describes the segmentation criterion (if any) that was applied to discover the rule. Segcomplex is nil if no sementation condition was used. This complex is interpreted using the `l4symboltable` according to the special rules for segmentation complexes mentioned in 3.2. Each normal variable in the `segcomplex` is interpreted as involving the corresponding delta variable.

`lookback`: This gives the value of the look back parameter for this rule. It tells how far back previous events must be consulted in order to use this rule. A rule with a `lookback` of 1 examines the previous card in order to predict the next card. For periodic rules, the `lookback` indicates how much look back takes place *within* a given *phase.*

`model`: Tells which rule model is used by this rule.

`nphases`: Tells, for periodic rules, how many phases make up the period.

`nextrule`: A link to the next rule in a linked list of rules (used for managing rules on the free list `fvl1r`

As mentioned above, rules developed in levels 1 and 2 are placed in the `vl1rulebase`. `Level4examine` inspects this rule base in order to obtain the results of the lower layer processing. The `vl1rulebase` is an array of records each containing the fields:

`rule`: a pointer to the rule at this entry.

`eoi`: The estimated value of the expected size of the set of legal cards according to this rule (unimplemented)

`minoi`: The minimum size of the set of legal cards (0 implies that this rule has a dead end) (unimplemented).

`maxoi`: The maximum size of the set of legal cards (unimplemented).

`stringsok`: A boolean flag. If true, then this rule is consistent with negative string plays. (i.e. at least one card in each negative string play is incorrect according to the rule).

Most of these fields are computed by `level4examine` while it is attempting to decide if a given rule is plausible.

3.4 VL_2 Modifications.

The representation of VL_2 entities (rules and complexes) is very similar to that of the VL_1 entities which have been described so far. This section details the differences between the VL_2 representations and the VL_1 representations.

3.4.1 VL_2 Complexes.

There is no such thing as a VL_2 layout. All events in layouts are VL_1 complexes. VL_2 complexes are used, however, in VL_2 rules. Conceptually, a VL_2 complex is a conjunction of VL_2 selectors. The `vl2complex` data structure is basically an array of selectors (the conjunction is implicit). The `vl2complex` also contains an `nselectors` field which tells how many selectors are in the complex. (This information is implicit in `vl1complexes`. Each VL_2 selector is quite complicated since it permits functions and operators in the reference. These reference functions permit the representation of sum and difference variables in a natural and convenient way. The fields of the `vl2selector` are:

`lhsfunction`: The `vl2symboltable` entry for the function in the referee of the selector.

`lhsdummies`: An array of `vl2symboltable` indices for the dummy variables of the `lhsfunction`. Presently the program can only understand unary functions (although it can parse more complex functions).

`rhsfunction`: The `vl2symboltable` entry for the function in the reference of the selector. (zero if no function appears in the reference).

`rhsdummies`: dummy variables for `rhsfunction`.

`rhsneg`: a boolean flag which indicates that the `rhsfunction` is to be considered to be negative. If true, then the selector is representing a "sum" variable. This results in the somewhat awkward description of $[value_0 + value_1 \geq 26]$ as $[value(card_0) = -value(card_1) + 26]$.

`relation`: as in the `vl1selector` this indicates the relationship which holds between the reference and the referee.

`operation`: This is the operator that appears in the reference. If `rhsfunction = 0` then `operation` should be `noop`. The operator `both` stands for $\pm$ and is used for delta variables of the form $[dvalue_{01} = -3..3]$ (so that we get $[value(card_0) = value(card_1) + -0..3]$).

`reference`: This is again a PASCAL set representing the elements in the reference. For sum descriptors, reference values may be much larger than `domain.max` in the `vl2symboltable`. (e.g. $[value(card_0) = -value(card_1) + 26]$ the 26 is in the `reference` as an absolute value.). For sum and difference selectors, the values in the references are the actual values to be printed. Otherwise, `domain.bias` must be added to each reference value.

`useprintref`: is true if `printref` should be used to print the reference.

`printref`: As with `vl1selectors`, intervals (both 2-sided and one-sided) are stored in `printref` and interpreted in combination with the `relation`.

3.4.2 VL_2 Symbol Table.

The `vl2symboltable` is only slightly different from the `vl1symboltables`. In addition to the `name`, `domain`, and `advice` fields, each `vl2variable` also has:

`dummyvar`: a field which indicates if this variable is a dummy variable (e.g. `card0`, `card1`, `string3`, etc.).

`subscript`: gives the subscript of the dummy variable (e.g. `card1` has subscript 1).

`cardorstring`: tells whether this dummy variable is a card dummy or a string dummy. This information is redundant since we could have used the `cors` field.

The dummy variables are initialized in `initsymbols`. Two auxiliary arrays are used to locate a dummy variable given its subscript. `Carddummies[i]` gives the `vl2symboltable` index of `cardi`. `Stringdummies[i]` gives the same for `stringi`.

3.4.3 VL_2 Rules.

`Vl2rules` are identical in structure to `vl1rules` except that `vl2rules` refer to `vl2complexes`. The `segcomplex` is a proper `vl2complex` rather than a special `vl1complex`.

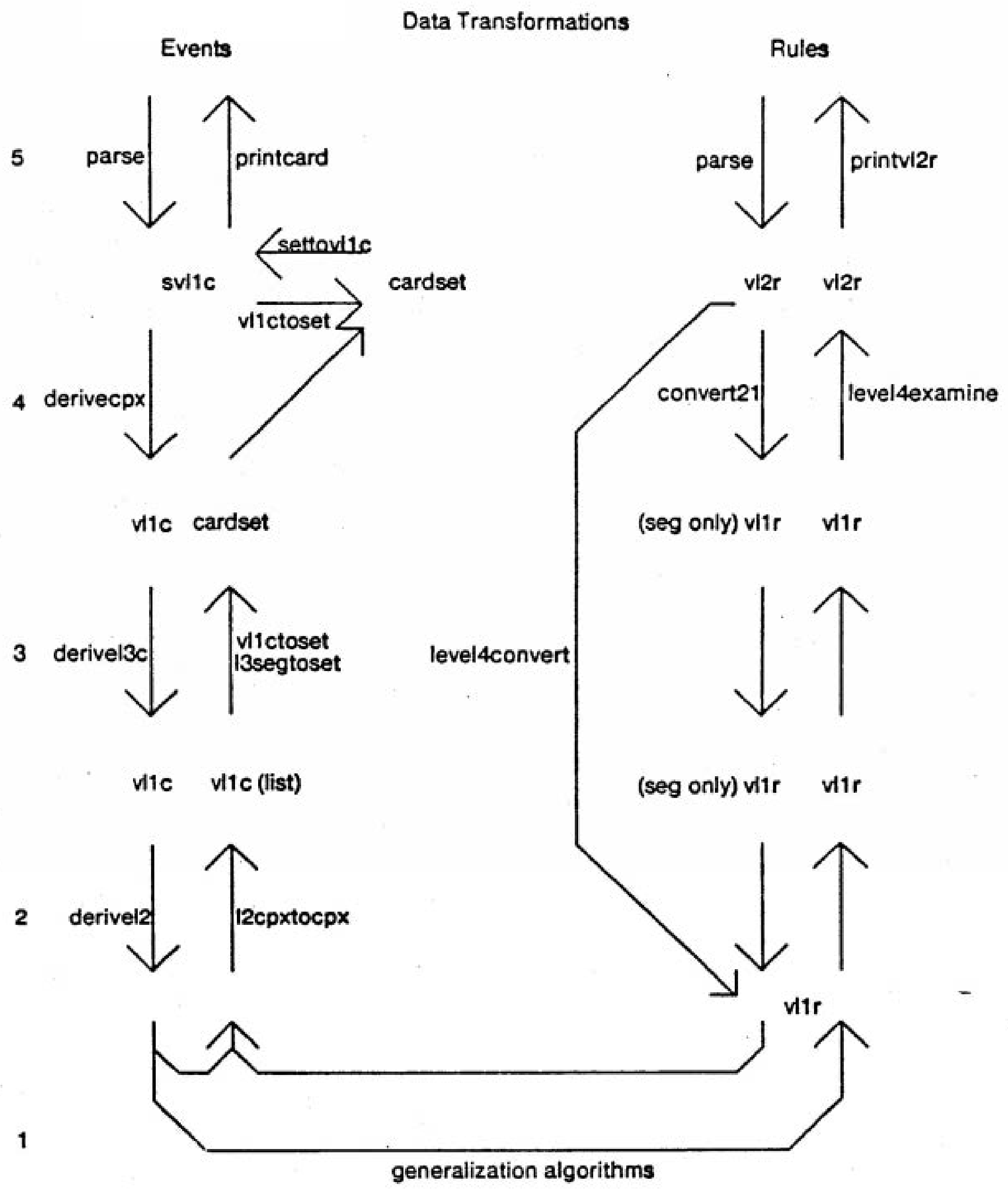

Data Transformations
Events
Rules
5
parse
printcard
parse
printvl2r
settovl1c
svl1c
cardset
vl1ctoset
vl2r
vl2r
4
derivecpx
convert21
level4examine
vl1c
cardset
(seg only) vl1r
vl1r
3
derivel3c
vl1ctoset
l3segtoset
level4convert
vl1c
vl1c (list)
(seg only) vl1r
vl1r
2
derivel2
l2cpxtocpx
vl1r
1
generalization algorithms

Figure 4-1

4. Data Transformations and Flow of Control.

Figure 4-1 provides a flow diagram of the various data transformations that occur in the program. By the end of this section you should understand this figure and the purpose behind each transformation. In this section, we examine each of the three major tasks (LE, PE, and CR) and trace their flow through the five layers of the system.

4.1 The Learning Element (LE).

The Learning Element (LE) has responsibility for examining the cards in the layout and proposing plausible rules to describe the layout. The LE is invoked by giving the top level command `INDUCE`. The rules which are proposed by the LE are placed in an array called the `vl2rulebase` where they may be manipulated by other commands (e.g. `LIST RULES, KILL, PLAY`).

4.1.1 User to Level 5.

The layout is entered into the program using the `CARD` command. Each `CARD` command corresponds to one player's turn in Eleusis and appends a string of up to four cards to the layout (along with the dealer's judgment concerning the cards). Cards are typed by the user as two-character combinations, e.g. `2C` means "two of clubs" and `QS` means "queen of spades." The lexical analyzer has a card token class, `sycard`. It converts `2C` into a `sycard` token with attributes of `suit=0` and `value=2`. The parser then converts this token into a simple VL_1 complex `svl1c` at semantic actions 14 and 15. The parser adds the `svl1c` to the layout at semantic action 6.

4.1.2 Level 5 to Level 4.

When the `INDUCE` command is given, the parser invokes the `level4le` to begin the rule discovery process. `Level4le` invokes `derivedescriptors` to convert both the `vl2symboltable` and the `layout` to the analogous level 4 structures: `l4symboltable` and `l4layout`. The procedure `copysymboltableinfo` copies relevant symbol table information from level 5 by copying all but the dummy variables. The level 4 and level 5 symbol tables are carefully initialized so that all `builtin` variables (see `advice.gen` above) are in fixed positions in both tables. All `builtin` descriptors must preceed all dummy variables in the `vl2symboltable`. Any descriptor defined in `vl2symboltable` (via the `DEFINE` command) will be copied to lower levels and used in the rule discovery process. Once a descriptor has been defined, it cannot be erased.

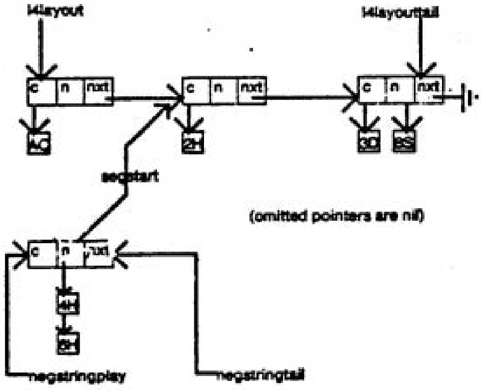
The level 4 procedure `derivecpx` is called to convert each `svl1c` in the `layout` to a full `vl1complex` in the `l4layout`. The conversion involves deriving all of the derived descriptors (e.g. *color, valuemod2*). While each simple `vl1complex` had selectors for *suit* and *value* only, level 4 `vl1complexes` have several selectors. The conversion is accomplished by converting each `svl1c` into a `cardset` containing one bit corresponding to that card. Then the `refsyms` entries are used to detect which values of which level 4 variables apply to that particular card. This derivation is performed for all variables which have `bitset` present in their `advice.gen` and `domain.cors = forcard`.

As noted in the thesis, strings of cards judged incorrect by the dealer (so-called negative string plays) are not used in the lower layers 1, 2, or 3. They are removed from `14layout` by `extractnegstrings` and saved in a special negative string layout whose head is `negstringplays` and whose tail is `negstringtail`. This layout is special in that it involves only negative examples. The `segstart` pointer in each `tsevent` points to the corresponding element in the `14layout` from which the negative string play was extracted.

For example, suppose we have the layout:

```
AC  2H   3D
    (4H  8S
    6H)
```

Where the 4H-6H were played was a string (and pronounced incorrect by the dealer). The `14layout` and `negstringplays` layouts will have the structure:



Where the `segstart` pointer points at the 2H event in `14layout`.

One other task of `level4le` is to convert the list of segmentation criteria from VL_2 to VL_1. Each criterion must be converted to a `vl1complex` in which normal variables are interpreted as delta (difference) variables (see above). The converted complexes form a linked list whose head is `l3segadvice`.

`Level4le` then calls `level3le` to perform the remainder of the LE task. The portion of level 4 which evaluates the generated rules and converts them back to VL_2 is procedure `level4examine`. If all symbol information were stored with the `vl1complexes`, a separate `level4examine` procedure would be unnecessary. However, since `vl1symboltables` are used to store symbol information, and since all `vl1rules` are described by the `l2symboltable`, and since the `l2symboltable` is rebuilt for each separate segmentation criterion, therefore it is necessary to copy the `vl1rules` and convert them to VL_2 after each call to `level2le`. This is a procedural kludge, but it works well. Conceptually, `level4convert` is still a part of level 4.

4.1.3 Level 4 to Level 3.

`Level3le` searches the space of possible segmentation conditions (as specified on `l3segadvice`) looking for plausible segmentations. For each segmentation, it builds a new `l3symboltable`, a new `l3layout`, and calls `level2le` and `level4examine`. `Level3le` also performs these steps without segmenting the layout (i.e. using the `nil` segmentation criterion). When a non-`nil` segmentation condition is being used, all string-related variables (e.g. *length*) are placed in the `l3symboltable` (see procedure `build32symbols`). Also all variables whose values remain constant under the segmentation are also installed in the `l3symboltable`. The `refsyms` array is used (inappropriately at this level) to deduce which variables do remain invariant under the given segmentation. For example, if the segmentation criterion is $[suit(card_0) = suit(card_1)]$, then *color* is also constant under this criterion. Thus, we can speak of the $suit(string_0)$ or the $color(string_1)$ as well as of the $length(string_0)$. Although *suit* and *color* are added to the `l3symboltable` and are then applied to strings of cards, the value of `domain.cors` remains `forcard`. This is important for converting back to VL_2 later.

If the program were strictly following the knowledge layer methodology, a domain independent symbolic reasoning strategy would be needed to deduce which variables remain invariant under the segmentation condition. This could be accomplished by applying theorem-proving techniques to the actual `DEFINE` commands and resolving against the segmentation condition.

A procedure `segmentlayout` derives the `l3layout` from the `l4layout` by finding maximal strings of contiguous cards which satisfy the segmentation criterion. This involves some subtlety when deriving negative events. For instance, a negative event could have violated the dealer's rule either because it violated the segmentation condition or because it violated some constraint on the relationship between segments. For example, in the layout:

AH	2C	2H	3D	3D	3H	4H
	3S	4S			3C	

When the rule is:

$$string = [value(card_0) = value(card_1)]: \qquad (A)$$

$$\begin{matrix}[length(string_0) = length(string_1) + 1] \\ [value(string_0) = value(string_1) + 1]\end{matrix} \qquad (B)$$

The 3S violates the (B) part of the rule, the 4S violates the (A) part of the rule, and the 3C violates the (B) part of the rule. Now, below level 3 the program is only performing induction on the (B) part of the rule. Thus, violations of the (A) part of the rule do not provide useful negative examples (e.g. The sequence AH 2C 2H is a legal sequence, not a negative example of a string that was too short). Violations of the (B) part of the rule do provide useful negative examples. However, we do not know that 3S violates part (B) of the rule because at segmentation time we do not *know* part (B). Thus, the only negative card which can generate a negative string event is a card which is legal according the part (A), the segmentation condition, but was declared illegal by the dealer. This card must, by implication, be illegal according to part (B) and thus provides reliable negative evidence. In this case, the 3C is such a card. The negative event $[length = 4][value = 3]\ldots$ is generated.

When a `nil` segmentation is applied, level 3 processing is trivial. The `l3layout` pointers are just copied from level 4 and the `l3symboltable` is also directly copied from the `l4symboltable`.

4.1.4 Level 3 to Level 2.

`Level2le` is called by `level3le` to continue the learning process. Level 2 is responsible for removing order from the events in the layout. Each layout represents order by a linked list. Level 2 retains this information in the form of delta (difference) and sum variables.

The `l2symboltable` is generated by a call to `genl2symboltable` which generates a symbol table entry for:

- each variable in `l3symboltable` with each of the possible subscripts $0, 1, 2, \ldots, maxlookback$.
- each variable in `l3symboltable` whose `advice.gen` contains `difference`. Differences are created between cards 0 and 1, 0 and 2, 0 and 3, and so on. Note that no differences are created between 1 and 2. There is no reason why such variables could not also be added.
- each variable in `l3symboltable` whose `advice.gen` contains `sum`. Sum variables are created summing cards 0 and 1, 0 and 2, 0 and 3, and so on. As with differences, this is an arbitrary choice which could easily be modified.

The order in which the variables are generated in `genl2symboltable` and the order in which variables are derived in `derivel2` must be identical so that the one-to-one correspondence between the symbol table and the VL_1 complexes is maintained.

`Deriveevents` derives an `eventset` which is simply two linked lists of events. The first linked list (`[0]`) comprises all negative examples. The second list of events (`[1]`) comprises all positive examples. For `DNF` and `DECOMP` rules, the `eventset` is the zeroeth element of the global array `F`. (i.e. `F[0,0]` is a linked list of negative events, and `F[0,1]` is a linked list of positive events.)

The `deriveevents` procedure has a local variable, `F`, which is just a single `eventset`, not an array of `eventsets` like the global `F`. This confusing notation was chosen in order to permit the use of the traditional notations `F[0]` for the negative events and `F[1]` for the positive events within `deriveevents`.

The other entries of the global `F` are used for the different phases of a `PERIODIC` rule. One `eventset` is developed for each phase. The event set for phase i is placed in global `F[i]`.

Level 2 performs some manipulation of the `l3layout` by shifting all of the negative events (`negcomplex`) right one `tsevent`. This makes the task of `deriveevents` much easier and also makes `splitlayout` easier. `Splitlayout` breaks the layout into several separate layouts, one for each phase of a `PERIODIC` rule. Before level 2 exits, it must un-split and un-shift the `l3layout` in order to restore things to their original configuration.

In addition to the `l2symboltable`, level 2 makes use of some additional symbol table information: the `l2subscript` array. `L2subscript` is an array exactly parallel to `l2symboltable`. They are both indexed by `vl1varindexes`. Conceptually, it adds two extra fields to the `l2symboltable`:

`l2subscript[varindex, 0]` gives the first subscript of this variable.

`l2subscript[varindex, i]` gives the second subscript of this variable (zero if none).

`Level2le` uses one symbol table for all of its calls to `level1le`, but it derives different event sets for each model. For the decomposition model, all variables defined in the `l2symboltable` are derived and passed to level 1. For the DNF and periodic models, only the variables involving $card_0$ are derived (e.g. $color_0$, $dsuit_{01}$, but not $card_1$).

4.1.5 Level 1.

The `level1le` is basically a switch to one of three separate routines. `DNF` rules are proposed by the `aq` procedure, `DECOMP` rules are proposed by the `bestdecomp` procedure, and `PERIODIC` rules are proposed merely by uniting the positive events in each phase by calling `unitephases`.

After the specific algorithms have been applied, a sort of general-purpose adjustment procedure is used to post-process the rules and detect symmetry. The rules are then added to the `vl1rulebase`. These level 1 routines, especially the post-processing routines, could be improved by performing further experimentation and modification.

4.1.5.1 `Aq`

The procedure `aq` implements the A^q algorithm. A good explanation of the algorithm is contained in the appendix to Larson's PhD thesis. `Aq` makes use of the `aqstar` procedure which develops an approximation to the set of all prime implicants which cover a given event, *el*. `Aqstar` operates by selectively computing the complement of the set of negative events, `F[0]`. The procedure `extend` implements the "extension against" generalization rule. The procedure `aqtrim` calls a general functional sort procedure, `trim`, to select the best N elements in a linked list of complexes according to the cost functions defined in procedure `aqcost`.

`Aq` makes use of some special data structures. First, each `vl1complex` contains two boolean flags, `fp` and `fq`. If a `vl1complex` is in the event set `F[1]`, then if `fp` is true, the `vl1complex` has not yet been covered by any star. If `fq` is true, the `vl1complex` has not yet been covered by any complex on the `mq` list. The set of complexes with `fp` true is always a subset of the set of complexes with `fq` true. (In other words, more events are covered by stars than by selected complexes on `mq`).

In order to develop complexes which are disjoint, `aq` adds each solution complex to the `F[0]` list. In order to reverse this modification to `F[0]`, a pointer, `f0start` is maintained which points to the "true" head of the `F[0]` list. The other elements that preceed `f0start` are released at the end of `aq`.

4.1.5.2 `Bestdecomp`

The `Bestdecomp` algorithm is quite complex, and even the description in the thesis is incomplete.

`Bestdecomp` uses several special data structures. The most important of these is the `coverrec`. Recall that a decomposition description is a set of "if–then" rules. The "if" parts of these rules are all disjoint complexes which exhaust the space of possibilities. The entire description forms a decision tree for determining, given the previous cards in the sequence, what cards are now playable. Each `coverrec` is a node in such a decision tree. For example, the representation of the rule:

$$
\begin{aligned}
[color_1 = red][value_1 > 6] &\Rightarrow [suit_0 = club] \\
[color_1 = red][value_1 < 7] &\Rightarrow [suit_0 = diamond] \\
[color_1 = black][value_1 > 6] &\Rightarrow [suit_0 = heart] \\
[color_1 = black][value_1 < 7] &\Rightarrow [suit_0 = spade]
\end{aligned}
$$

is shown in Figure 4-2 where each box represents a `coverrec` and the leaves of the decision tree are `vl1complexes` describing the right-hand sides ("then" parts)

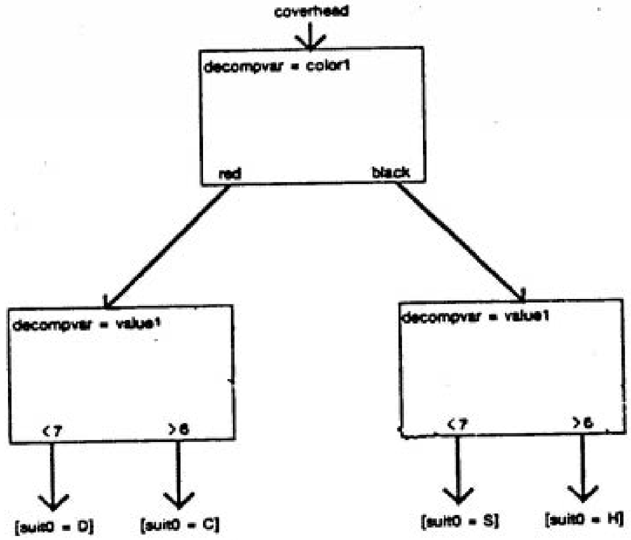


Figure 4-2

of the "if–then" rules. The global variable `coverhead` is the root of the cover which is under construction by `bestdecomp`.

Each `coverrec` has the fields:

`nextcover`: a link to the next `coverrec` when `coverrecs` are on the `fcover` free list.

`decompvar`: the `l2symboltable` index of the variable which is to be tested at this node. In the topmost `coverrec` of Figure 4-2 this points to the `l2symboltable` entry for *color*$_1$.

`leaf`: a variant tag which indicates whether or not this is a leaf of the tree. If this is a leaf in the decision tree, all descendants of this node are `vl1complexes` which describe the right-hand sides of the rules. If this is not a leaf, then the descendants of this node are also `coverrecs` which test the values of some other left-hand side variable.

`ccpx`: an array indexed by reference values. Each reference value indexes into `ccpx` to get a `vl1complex` corresponding to that value.

`clist`: also an array indexed by reference values. Each reference value indexes into `clist` to get a new `coverrec` corresponding to that value.

How does the program handle cases like [$value_1 < 7$]? This is done by setting `nbranches` to a non-zero value and placing the `vl1complex` [$value_1 < 7$] in the `branchconditions` array:

`nbranches`: indicates the number of branches in the `branchconditions` array at this node in the decision tree. If `nbranches` is zero, then no `branchconditions` are used and the conditions correspond to the simple reference values for the `decompvar`.

`branchconditions`: provides a `vl1complex` describing the condition connected to this branch in the decision tree. The values used to index into the `branchconditions` array are used to index into `ccpx` or `clist` to obtain the corresponding branches in the tree. Notice that `maxbranch` must be less than or equal to `maxvalue` for this scheme to work. (But it always is since branch conditions are generalizations of reference values).

Figure 4-3 shows the same tree as Figure 4-2, but with the values of these fields

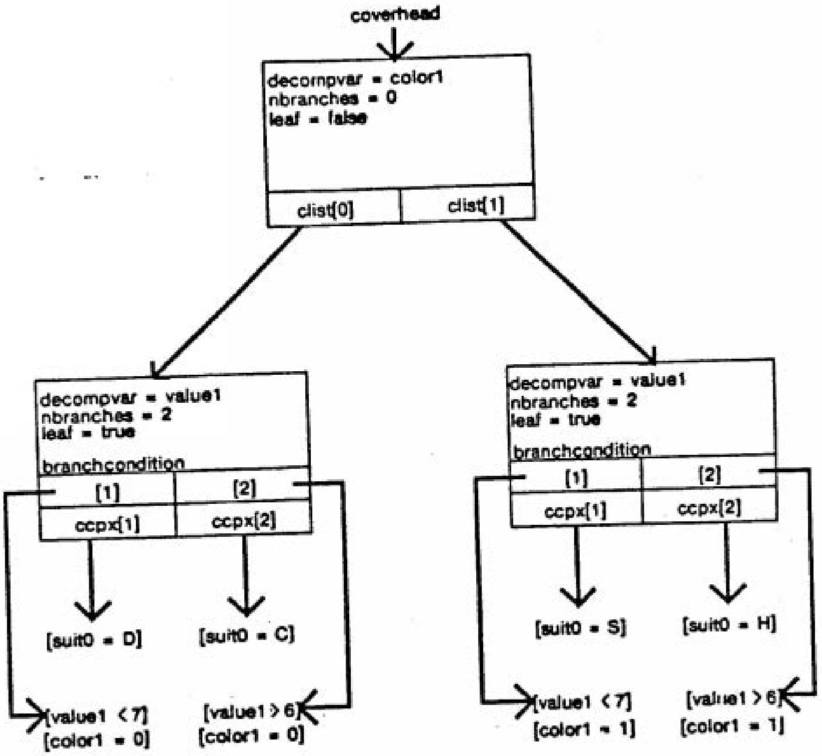


Figure 4-3.

added.

The decomposition algorithm builds the decision tree `coverhead` using the information in the array `decomptab`. `Decomptab` is an array parallel to `12symboltable` so that each variable in `12symboltable` has an entry in `decomptab`. However, not all of these entries are used. Only those entries corresponding to "left-hand side" variables (e.g. $color_1$, $value_1$, but not $suit_0$ or $dcolor_{01}$) are used. From each of these elements, trial decompositions are built and evaluated by the decomposition algorithm.

In order to determine which variables are left-hand side variables, the array `lhsvar` (built by `level21e`) is consulted. `Lhsvar[i]` is true if and only if variable i is a left-hand side variable.

Each element of `decomptab` contains the fields:

`cover`: a pointer to a `coverrec` which is a trial decomposition based on this variable. In other words, `decomtab[i].cover↑.decompvar = i`.

`cost`: an array giving the evaluated values of the cost functions as applied to this variable (see function `decompcost`)

`reallydone`: true if this variable has already been selected for decomposition in a prior iteration (and therefore already has a `coverrec` in the `coverhead` decision tree.

`done`: true if `reallydone or not lhsvar[i]`. If a variable is marked `done`, there is no need to develop a trial decomposition for it.

The algorithm operates as follows (see `bestdecomp` body). First, `initcovers` is called to place a `coverrec` under each `lhsvar` in the `decomptab`. These `coverrecs` have `leaf = true` and `nil ccpx` entries. `Coverhead` is set to `nil`.

Next, we repeatedly develop trial decompositions and select the best decomposition until a complete and consistent cover has been found. Each iteration involves:

1. Using `cleantab` to set the `done` flag properly.
2. Using `traverse` to build a trial decomposition under each `decomptab` variable that is `not done`. `Traverse` recursively traverses the `coverhead` decision tree in depth-first, left-right order (preorder). At each leaf of the tree, it calls `buildcovers` to build a new trial decomposition under each `decomptab` variable. Thus, if `coverhead=nil`, `buildcovers` is called once. If `coverhead` has the configuration shown in Figure 4-4, then `buildcovers` is called twice, once for each value of $color_1$. The resulting configuration of `decomptab` is shown in Figure 4-5. Note that each `ccpx` entry in these trial decompositions contains a linked list of `vl1complexes`. One complex for each `nil` leaf node in the `coverhead` tree. Each `vl1complex` is the union of all positive events which meet the particular left-hand side conditions at that point in the tree. The `vl1complex` marked with a * in Figure 4-5 is the union of all positive events for which $color_1 = 0$ and $value_1 = 0$. If there are no positive events which satisfy the conditions at one of these nodes, then the `vl1complex` will have all empty references (see `cleancpx`).

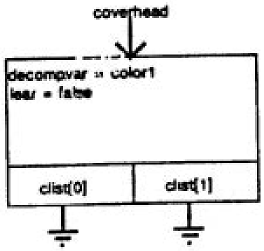


Figure 4-4.

3. `Buildcovers` (as called by `traverse`) invokes `mergeunions` to apply the rules of generalization to these left-hand side variables. The procedure `mergeunions` is very messy and involved. It is discussed below. The result after `mergeunions` is shown in Figure 4-6. Now all of the `vl1complexes` for [$value_1 = 0..6$] have been merged into a single complex (and so also for [$value_1 = 7..12$]).
4. If `decgen` (the decomposition generalization parameter) is 1 or 2, then `genrlcovers` is called to generalize the references of the complexes hanging off `decomptab`. If `decgen=2`, overlapping selectors are removed from these trial decompositions.

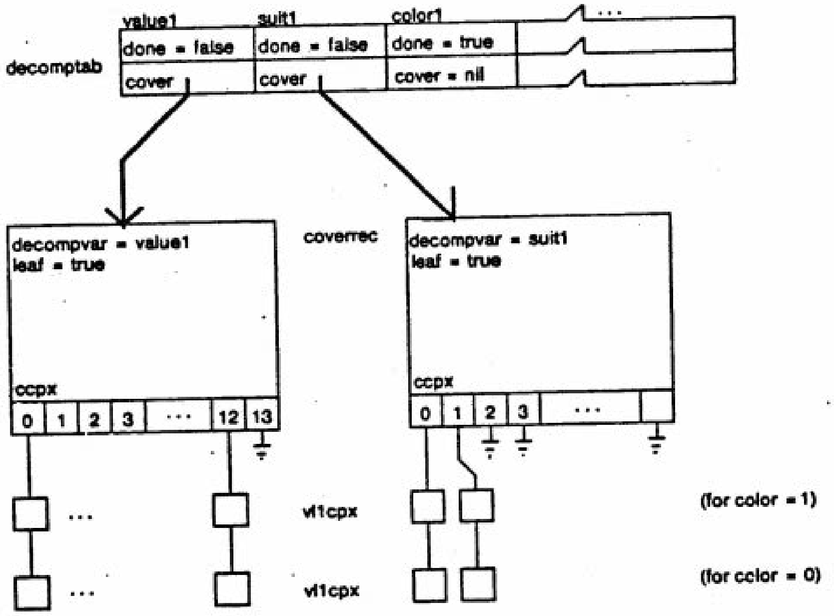


Figure 4-5.

5. `Selectbest` is invoked to select the best trial decomposition to add to the `coverhead` decomposition tree. `Selectbest` invokes the generalizaed functional sort procedure `trim` to determine the best variable subject to the function `decompcost`. Note that `decompcost` code 1 determines if this trial decomposition covers any negative events. If no negative events are covered, the decomposition is "consistent". If the selected decomposition is consistent, then the `coverfound` boolean flag is set to true. This terminates the algorithm.

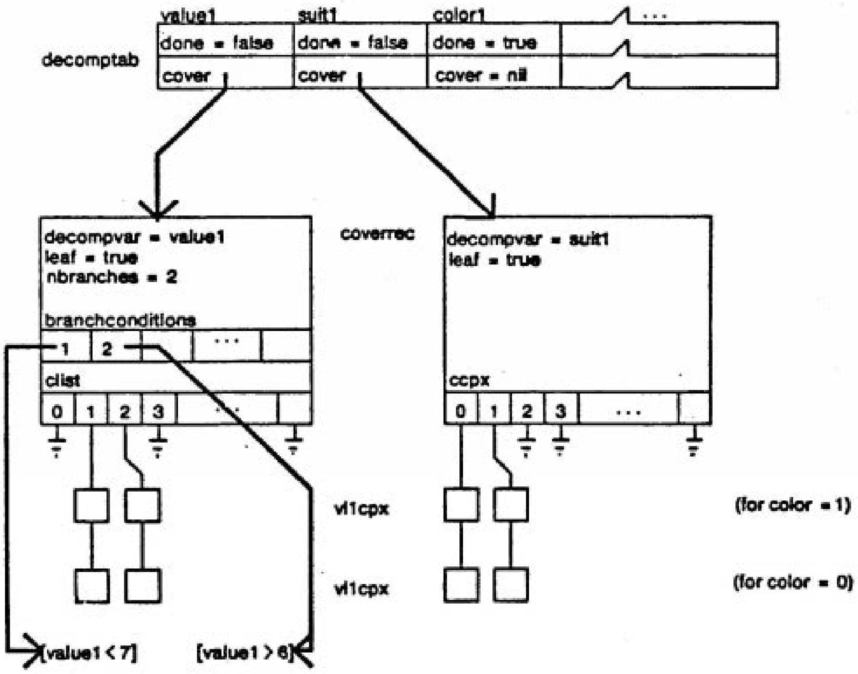


Figure 4-6

6. `Unwind` is called to traverse the `coverhead` decision tree while unwinding the selected decomposition. One "row" of `ccpx` complexes is unwound for each leaf in the decision tree so that we get the tree described in Figure 4-2.
7. `Freeallvars` is called to release all of the unselected trial decompositions. The `ccpx` lists are released and reinitialized to `nil`. Note that the `coverrecs` remain.

As indicated above, this iteration continues until the cover is consistent with the negative events. It is possible that the cover will never be consistent, e.g. setting `decgen=2` can cause all of the trial decompositions to become inconsistent. The loop will also terminate after `maxcomplex` iterations. This is a slight misuse of the `maxcomplex` parameter. In all other contexts, `maxcomplex` is the maximum number of complexes in a rule. A possible improvement would be to change `bestdecomp` so that that interpretation applies here as well.

Once the `coverhead` tree has been generated, it is returned as a simple linked list of complexes. The procedure `pullresult` traverses the tree and builds this linked list.

One subtlety of the decomposition algorithm involves how the left-hand side condition is integrated with the `vlicomplexes` hanging off the tree. `Traverse` maintains a `condition` `vlicomplex` which describes the present left-hand side condition ("if" part) of the rule. This is passed to `buildcovers` where it is used to initialize the new `vlicomplex` which will cover that particular branch of the trial decomposition. `Mergeunions` also maintains this integration. Consequently, `pullresult` does not need to concern itself with the left-hand side variables at all—they are already set correctly in the `vlicomplexes` in the `coverhead` tree.

The process of applying the rules of generalization to the trial decompositions in the `decomptab` is very involved. Reread the description of `mergeunions` on pages 21 and 22 of the thesis. The algorithm operates by computing the distance between adjacent values of the left-hand side variable under consideration. When two adjacent values are relatively distant from one another, the program tries to break the domain at that point and create two or three "if" conditions which generalize the original "if" conditions.

For each left-hand side variable which has more than `maxcomplex` values in its domain, `mergeunions` goes through the following steps:

Step 1: Initialize Relevancy Coefficients. The array `rc` is an array of relevancy coefficients in parallel with `12symboltable`. These numbers, referred to in the thesis as "weights," are used to compute the weighted Hamming distances between complexes. In this step, the values of `rc` are initialized by copying the `advice.plausibility` field of each variable.

Step 2: Initialize the Generalized Cover. The local variable `newcov` contains a `coverrec` which will replace the trial decomposition for the variable we are analyzing. In this step, we initialize `newcov` by copying the complexes hanging off the `decomptab` entry for this variable and generalizing their references by calling `generalizereferences`.

Step 3: Adjust Relevancy Coefficients according to the discriminating ability of the variables. For instance, if a selector has the value * (irrelevant) in any of the complexes in `newcov`, then the `rc` of the variable in that selector is set to 0. If a selector intersects a selector in a negative example, then the `rc` of that variable is divided by a factor of 4. If all selectors for a given variable have the same value, then the `rc` is set to zero. This last test is accomplished by computing the union of all of the references of all of the complexes in `newcov`. This union is created in the array `grandref`. If a reference in `grandref` has only one element in it, then all of those selectors had the same value.

Step 4: Compute Distances between adjacent values of this variable. This process is complicated by the fact that not all values of the variable of interest will have actual events with that value. For example, in Table 2 of the thesis, the variable of interest is $value_1$. Events only existed for values A, 2, 3, 6, 10, and J. In the source code, missing values such as 4 or K which have no events are called `[]-cases`. Distances are only computed between `non-[]` cases. For each pair of adjacent `non-[]` cases, a distance is computed. Cases are adjacent if they can be generalized using the "closing interval" generalization rule. The array `distance` is used to keep track of the distances as they are computed. Each element in `distance` is a record with fields `left`, `right`, and `d`. `Left` and `right` are domain values which indicate the `non-[]` cases between which the distance, `d`, is being computed. The function `dist` actually performs the distance computation.

Step 5: Locate Maxima. In this step, local maxima in the distance array are identified and stored in the `maxima` array. `Maxima` is an array of domain values. Each value is actually an index into `distance` which gives the local maximum. `Nmax` is the number of maxima which are found. An additional step 5.5 was inserted here to remove minor maxima (maxima which are smaller than one tenth the size of the absolute maximum distance).

Step 6: Determine how the domain should be broken into subintervals. If no maxima, or more than two maxima were found, we give up on this variable and try another. For most linear-type domains, if there is one maximum, the domain is broken into two cases. If there are two maxima, the domain is split into two or three cases depending on the "end-around distance" (i.e. the distance between the largest and smallest `non-[]` cases). Instructions which tell which values of the domain should be merged together into a single case are placed in the `merge` array. Subfields of `merge` are `left` (the smallest value in the interval of values to be merged), `right` (the largest value in the interval), and `reference` (a bit set indicating the actual set of domain values to be merged. Once the merge values have been determined, `branchconditions` for `newcov` are built.

Step 7: Merge old branches into new branches. Procedure `unite` is used to unionize the complexes hanging off of `newcov`. Finally, `newcov` is put into its proper place in the `decomptab`.

There is something very unsatisfying about procedure `mergeunions`!

4.1.5.3 Periodic.

The discovery of `PERIODIC` rules is the simplest of the three models. The procedure `unitephases` creates one `vl1complex` for each phase using the positive events in global `F[1..nphases, 1]`. The resulting complexes are linked together (with phase 1 at the head of the linked list) and returned to `levelile`. The remainder of the processing is accomplished in the common adjustment code.

4.1.5.4 Common Adjustment Code.

`Adjustcomplexes` attempts to post-process the rules to give them more symmetry and remove redundant and irrelevant variables. This is a two-step process as described in the thesis.

First, `generalizereferences` is called to generalize the references of the selectors. Then overlapping selectors are removed. The complex `united` is the union of all complexes in the rule. If a complex intersects `united` before that complex has been combined with `united`, then there are selectors which overlap. The information concerning overlapping selectors is contained in the array `kill`. `Kill` tells for each variable if its corresponding selector should be deleted (made irrelevant).

If the first step of generalizing selectors leads to inconsistency, then the `aqstar` procedure is invoked to try to extend the un-generalized complexes against the corresponding negative events. This call to `aqstar` is a side effect of the function `consistent` which ascertains whether or not a rule is consistent with the negative examples. This step has not worked well. In fact, it is possible to get a rule which is consistent with the negative examples but does not cover all of the positive examples. This is another problem that needs to be solved in this program.

If the second step leads to inconsistency, the results of the first step are returned as the answer.

Notice that there is great similarity between `adjustcomplexes` and `genrlcovers`. It is likely that DECOMP rules do not need to be post-processed by `adjustcomplexes`.

4.1.6 Level 1 to Level 2.

After `level1le` has discovered rules, level 2 has an opportunity to process these rules and remove rules that are implausible, etc. The procedure `cleanrules` was intended to perform this task, but it was never written.

An idea for `cleanrules`: often rules like

$$[color_1 = red] \rightarrow [face_0 = true][value_0 > 10]$$

are induced by level 1. The $[value_0 > 10]$ is redundant and could be removed. This is a tricky problem because the combination

$$[face_0 = true][value_0 > 11]$$

is different. Here, the $[face_0 = true]$ should be removed.

Some general rules to solve this problem should be formulated and implemented.

One particularly appropriate case is the combination of a delta variable and a normal variable. For example in Example 1 in the thesis, rule 1 has the joint occurrence of $[value(card_0) >= Jack][value(card_0) > value(card_1)]$. The second, delta descriptor is redundant in this case (actually both of them are redundant, but that's an example of the above equivalence problem). Since level 2 has knowledge of delta and sum descriptors, it is an appropriate place to detect and solve this problem.

4.1.7 Level 2 to Level 3.

Presently, the `level3le` performs no post processing on newly discovered rules. As noted in the thesis, this is bad. It is presently possible to discover a segmented rule where the rule is inconsistent with the tail end of the layout. For instance, the program will discover the rule:

$$string = [value(card_0) = value(card_1)]:$$

$$[length(string_0) = length(string_1) + 1]$$

to describe the layout

AC 2H 2D 3H 3S 3D 4H 4C 4D 4S 4C 4D 4H

because levels 1 and 2 never know about the string of 4's and level 3 does not check to see that the discovered rule is consistent with the tail end of the layout. Some code like that of `l3checkseg` should be developed to do this. Or perhaps we could call `level2cr` instead.

4.1.8 Level 3 to Level 4.

Level 3 invokes `level4examine` to convert the rules in the `vl1rulebase` to VL_2. `Level4examine` performs Eleusis-specific tests of the rule and if it passes these tests, it is then added to the `vl2rulebase`.

The tests are:

1. Check that the expected value of the size of the set of legal cards (`EOI`) is within the limits [*mineoi*...*maxeoi*] and that the rule has no deadends.
2. Check that the rule is consistent with the `negstringplays`.

Only the second step is actually performed because the procedure `sizeup` was not implemented. I have notes on how to implement this routine if anyone wants to try.

The negative strings are checked by modifying the layout to make it appear as if the negative string had been played and judged correct by the dealer. Then `level3cr` is called to check that the present rule is inconsistent with the layout. If the rule is consistent with this modified layout, then it is a bad rule. A global flag, `dontconvert` is set to tell `level3cr` that the rule is already in VL_1 form and need not be converted from VL_2 as in the normal critic case.

4.2 The Critic (CR).

The critic portion of the program is responsible for checking a VL_2 rule to see that it is consistent with all of the evidence in the layout. The critic is invoked whenever the user enters a VL_2 rule and whenever the user gives the EVALUATE command. The EVALUATE command checks each rule in the VL_2 rulebase for consistency using the CR and then invokes the Performance Element to decide which cards can be played at this time according to this rule.

In order to evaluate a rule, both the rule and the layout must be converted to VL_1 representations. The conversion of the layout into particular sets of VL_1 events is identical to the conversion process used in the LE. The rule conversion process is described now.

4.2.1 User to Level 5

The user can enter a VL_2 rule into the rule base using the RULE command. The details of the parser which converts the user input into the `vl2rule` data structure are discussed in section 5.

4.2.2 Level 5 to Level 4.

The first step in converting a VL_2 rule to VL_1 is to convert the segmentation condition in the rule to VL_1 format. As in the LE, the segmentation condition is represented as a `vl1complex` in which normal variables at level 4 (`l4symboltable`) are interpreted as delta variables. This conversion process is accomplished in `convert21 (conseg21)`. The `level4cr` procedure cannot complete the rule conversion because the VL_1 representation of a rule uses the `l2symboltable` which does not exist yet. The remainder of the VL_2 rule (the list of `vl2complexes`) is placed in the global variable `l4complexes`. The remainder of the conversion process is accomplished when `level2cr` calls `level4convert`.

`Level4cr` also builds `vl1complexes` for each input event on the `layout` and saves the negative string plays as in the `level4le` procedure.

4.2.3 Level 4 to Level 3.

The Level 3 critic merely builds the `l3symboltable`, segments the layout according to the segmentation condition in the `vl1rule`, and calls the level 2 critic to continue the evaluation.

4.2.4 Level 3 to Level 2.

`Level2cr` creates the `l2symboltable` and then invokes `level4convert` to complete the conversion of the `vl2complexes` in the rule into `vl1complexes`.

The `l3layout` is converted into the appropriate event sets in the global array `F`, and then the `level1cr` is invoked.

4.2.5 Level 2 to Level 1 to Level 2.

The level 1 critic first checks to see that the rule is consistent with respect to the negative examples in `F`. Then it checks to see that the rule covers all of the positive examples in `F`. The boolean result of this operation is passed back to level 2.

4.2.6 Level 2 to Level 3.

Level 2 merely passes the boolean result of `level1cr` to level 3.

4.2.7 Level 3 to Level 4.

Presently level 3 does not check to see that the rule is consistent with the tail end of the `l4layout`. If the rule involves segmentation, this can lead to an incorrect evaluation of the rule as explained in 4.1.7.

4.2.8 Level 4 to Level 5.

The level 4 critic receives the result from level 3 and then checks to see that the rule is consistent with the negative string plays by calling `checknegstrings`. If it is consistent, then the value `true` is returned to level 5.

4.2.9 Level 5 to User.

The `vl2rulebase` can be printed using the `LIST RULES` command. The procedure `printvl2r` is called for each VL_2 rule in the `vl2rulebase`.

4.3 The Performance Element (PE).

The Performance Element is responsible for determining the set of cards which is currently playable according to each rule in the `vl2rulebase`. The operation of the PE is a combination of the LE (converting the layout to `vl1complexes`) and the CR (converting `vl2rules` to `vl1rules`) and some additional work all its own (developing some description of the set of legal cards playable according to the rule). The output of the PE is a `cardset` describing the set of legal cards according to a given rule. This `cardset` is compared with each card in a player's hand to determine which cards in the hand are playable according to the rule. This information can be displayed using the `LIST HAND` command.

4.3.1 User to Level 5.

The user types the layout as described in 4.1.1. The user also enters cards into his/her "hand" by the use of the `HAND` command. The `hand` is an array of `handelements`. Each `handelement` has the fields:

`card`: a `cardset` with one element in it.

`goodrules`: a set of indices into the `vl2rulebase`. A bit is set in `goodrules` corresponding to each rule in the `vl2rulebase` according to which this card is currently playable.

The proper setting of the `goodrules` field is really the end result of the PE.

4.3.2 Level 5 to Level 4.

`Level4pe` builds the `l4symboltable`, derives the `l4layout`, and extracts the negative string plays. It converts the segmentation condition of the `vl2rule` to the peculiar VL_1 delta representation as described in 4.2.2.

4.3.3 Level 4 to Level 3.

`Level3pe` builds the `l3symboltable` and derives the `l3layout` by segmenting the `l4layout`. It then calls `level2pe`.

4.3.4 Level 3 to Level 2.

`Level2pe` is the bottom level for the Performance Element. There is no `level1pe` because the whole concept of extending a sequence is inappropriate at level 1. `Level2pe` builds the `l2symboltable` and completes the conversion of the `vl2rule` to VL_1 by calling `level4convert`. Now comes the interesting part of the PE. The `level2pe` returns a disjunction of VL_1 complexes (represented as a linked list of `vl1complexes`) which describes the set of legal events which could continue the sequence. This is accomplished by taking each complex in the `vl1rule` and modifying it so that it properly describes the current set of legal events. The modifications involve three steps:

1. If the complex tests variables in previous events (e.g. $value_1$, or $suit_2$), then we check it against those events to see if it is currently applicable.
2. If the complex is applicable, then we set all selectors which do not involve the last event to have the value * (irrelevant).
3. Adjust the references of selectors which describe only the last event (e.g. $card_0$ or $string_0$) so that they properly reflect the operation of delta and sum variables. For example, if the last event had $[value = 5]$ and the rule says that $[dvalue_{01} > 0]$ then we adjust the $value_0$ selector so that $[value_0 > 5]$.

`Level2pe` calls `l2cpxtocpx` to perform these tasks. `Level2pe` must build a layout for `l2cpxtocpx` to use. This is trivial for `DECOMP` and `DNF` rules. For `PERIODIC` rules, only the layout corresponding to the proper phase is passed to `l2cpxtocpx`.

`L2cpxtocpx` uses `deriveevents` to derive the last event in the layout (complete with delta and sum variables). Then `lhscovers` is called to determine if the given complex is applicable. Then a new `vl1complex` is developed which has the proper reference values.

4.3.5 Level 2 to Level 3.

The `level3pe` has a difficult task when the rule involves segmentation. Consider a rule to be made up of two parts: the segmentation condition, S, and the rule body, B. There are two ways to play according to the rule:

Strategy 1: continue the current segment by playing according to S

Strategy 2: discontinue the current segment by playing according to B and *not* S.

We can use strategy 1 whenever B does not force us to change segments. We can use strategy 2 whenever B gives us the option (or forces us) to change segments. We are forced to change segments because of some limitation on *length* in part B of the rule.

The decision of which strategy to use depends upon the tail end of the layout. Recall that the tail end is an incomplete segment which was not passed to the lower levels. The length of the last segment is stored by `segmentlayout` in the global variable `lastlength`. `L3checkseg` determines if the current segment has a choice of continuing, must continue, must end, or is inconsistent with the rule. If the current segment must end or has a choice, then we call `level2pe` again, this time with the tail end of the layout added on as an extra event.

The complexes returned from `level2pe` are converted into a `cardset`, `rset`, representing the set of legal cards. The segmentation condition, S, is also converted into a `cardset`. The `cardset` is complemented and intersected with `rset` to give the final return value. This corresponds to strategy 2.

If we have the choice of continuing or if we must continue the current segment, then the segmentation complex is converted into a `cardset` and unioned with `rset` to obtain the final return value.

4.3.6 Level 3 to Level 4.

Level 4 simply passes the `cardset` up to level 5.

4.3.7 Level 4 to Level 5.

Level 5 processes the `cardset` against the cards in the `hand` and sets `goodrules` properly.

4.3.8 Level 5 to User.

The user can print out the `goodrules` sets by issuing the `LIST HAND` command. A matrix of cards by rules is printed with a "Y" indicating when a card on that row can be played according to the rule in that column.

The user can also ask the program to select a card to play by using the `PLAY` command. The code assumes that the user has already done an `EVALUATE`. If the present `strategy` is `conservative`, then the program selects the card in the hand which is legal according to the most rules. If the `strategy` is `discriminant`, the program selects the first card in the `hand` which is legal according to at least one-fourth and no more than three-fourths of the rules.

5. The User Interface (Level 5).

Level 5 is implemented as an LALR(1) parser with a finite state machine for the lexical analyzer.

5.1 The Lexical Analyzer: `Gettoken`.

`Gettoken` performs lexical analysis. Figure 5-1 shows the finite state transition diagram for each of the terminal symbols. Quite a bit of work goes into disambiguating `Q` from `QS` and `QUEER` (a variable name?). Characters are read via `getchar`. The global variable `buffer` contains the line most recently read from the user. `Getchar` also prints the prompt each time the parser is in state 0 or 19 and it must read a new line. When end of file is encountered on `CFILE`, `getchar` switches to the file `TTY`. The global variable `chx` remembers the position in `buffer` of the last character returned by `getchar`.

`Gettoken` executes its finite state code. Then, if it has an alphanumeric identifier, it tries to find a reserved word which matches it. The global array `id` provides a character string to token cross reference. Some takes have attributes such as a numeric value, a character string, or even a suit and value (in the case of cards). No state information (besides `chx` and `buffer`) is retained between successive calls to `gettoken`.

5.2 The Parser.

The parser is a table-driven parser produced with the aid of the program `yacc` on `UNIX`. The book, *Principles of Compiler Design*, by A. V. Aho and J. D. Ullman (Addison-Wesley: 1978) provides the theoretical background for LALR(1) parsing and `yacc`. The tables used by the parser (`acttab`, `actstart`, `gototab`, and `ntokens`) are built by taking the verbose output from `yacc` (the `-v` option) and massaging it with the text editor into PASCAL assignment statements.

`Acttab` is the action table. It tells, for a given state and a given input token, what to do next. The fields are:

- `t`: The token to match against the input token. The special token kind `sydefault` matches any input token.
- `s`: The semantic action to be executed when this action is performed (see 5.3 below).
- `act`: The action, one of `shift`, `reduce`, `error`, or `accept`. A `shift` action causes the parser to push the current state onto the `pstatestack` (a stack of parser states) and enter the new state indicated by the `nextstate` field. A `reduce` action causes the parser to pop several states off the `pstatestack` (a number of states equal to the number of symbols on the right-hand side of the grammar rule) and then use the exposed state on the `pstatestack` and the `gototab` to determine which state to goto. An `error` action indicates that a syntax error has been detected. This causes a message to be printed and the parser exits to global label 9998 which has the effect of reinitializing the parser and restarting it. Thus, any pending input is ignored and the program prompts for a new command. An `accept` action never occurs in this language because there is no way to generate the final symbol `syeof`. The `Q` command is used to exit the program. It performs a global jump to label 9999.

The `actstart` table indicates where in `acttab` to start looking for the actions for a specific state. All actions for a given state are contiguous in `acttab` and the last action always has a `sydefault` `t` field.

The `ntokens` table indicates how many nonterminals are on the right-hand side of each grammar rule.

The `gototab` has entries indicating, for a given exposed state on the `pstatestack`, what the next state to goto is.

The process for a reduce action is to use `acttab[].rule` to index into `ntokens` to find out how many states to pop off the `pstatestack`. Then the `acttab[].gotostart` field is used to index into `gototab`. The `gototab` is scanned until `gototab[].exposedstate` matches the top state on the `pstatestack`. Then the parser enters `gototab[].nextstate` state.

5.3 Semantics

All of the work at level 5 is performed in procedure `semantics`. This section presents some pointers for how to understand what is going on in this large procedure.

Procedure `semantics` is called whenever the action selected by the parser from `acttab` contains a non-zero `s` field. Almost all of these actions occur during `reduce` actions.

The semantic routines work with six global stacks:

`tokenstack(TS)`: This stack contains tokens which have been saved (for various reasons). In the commentary in the source code, the `tokenstack` is referred to as the `TS`. There is a special tie-in to the `tokenstack` in the parser. Whenever the parser shifts a token of class `syid`, `synumber`, `syvalue`, `syeq`, `syne`, `syle`, `syge`, `sylt`, `sygt`, `syplus`, `syminus`, `syboth`, or `syparameters` it also pushes that token onto `TS`. This is an efficiency move.

`v11cstack(1C)`: This is a stack of pointers to `v11complexes`. As cards are scanned by the parser, they are converted to `v11complexes` and pushed onto this stack.

`v12cstack(2C)`: This is a stack of pointers to `v12complexes`. As a RULE or SEG command is being parsed, `v12complexes` are pushed onto `2C`.

`v12vstack(2V)`: This is a stack of `v12variable` indices. It is used for parsing dummy variable lists.

`v12rstack(2R)`: This is a stack of pointers to `v12rules`. The `v12rule` under construction during a RULE command is kept here.

`countstack(C)`: This is a stack of integers. It is used for a variety of purposes. The most common use is to count things on other stacks. For instance, when `v12complexes` are being parsed, `C` contains the number of complexes on `2C`.

For each stack, there are push and pop procedures for manipulating the stack and there is a variable `top`*xxx* which indexes the top element of *xxx*`stack`. (e.g. `tokenstack[toptoken]` is the top element on the `tokenstack`.)

With each semantic action, there is a comment indicating which rule in the grammar is being reduced at this time. When a semantic action is called as a sideeffect of a `shift` action, a "." appears in the grammar rule indicating the present position of the parse. Each semantic action also indicates what it expects on each of the stacks and what it leaves on each of the stacks.

Some other auxilliary variables are kept.

`p1`: is a `costfunctional`. It is used to store a `costfunctional` under construction. In this sense it is like a separate, single-element stack.

`nextplay`: a global variable which indicates what number should be placed in the `play` field of the next string of cards (see semantic action 6).

`defining`: a global boolean flag which indicates if we are in the process of defining a new derived descriptor. `Defining` is used by actions 67 and 12 to signal `perror` to pull out the partial definition if an error occurs.

The procedure `perror` is the parser error message procedure. It prints an appropriate message, cleans the stacks and jumps to global label 9998 to restart the parse.

5.3.1 Design Problems with the Grammar.

There are a few problems with the present grammar that need attention. First of all, a more consistent syntax for the parameters is needed. As parameters were added, the names became awkward and inconsistent. Also, the way of specifying the parameters became quite unintuitive (e.g. `A SEGPLAUS 2 50` where the 2 is an absolute number and the 50 is a percentage).

Secondly, rules 72 and 73 in the grammar (see Appendix I of the thesis) should be extended to include `syvalue` as a valid `DEF` value. The code in the program will need to be changed slightly. The result will be to permit the same character string as a domain value in two different descriptors (This would be very useful with the string `TRUE`).

It would also be nice if the program could accept a selector of the form $[value(card_0) + value(card_1) < 26]$ rather than writing it in the awkward form $[value(card_0) < -value(card_1) + 26]$.

6. Miscellany.

6.1 Storage Allocation.

Since PASCAL does not guarantee that storage returned using the `DISPOSE` function is ever reusable, the Eleusis program manages its own storage. The routines `newvlic`, `freevlic`, and `freelvlic` are typical. `Newvlic` allocates a `vlicomplex` either by taking one off the `fvlic` free list or by calling `NEW`. `Freevlic` links a `vlicomplex` onto the `fvlic` list. `Freelvlic` is an efficient and convenient way to release a linked list of zero or more `vlicomplexes`.

In general, these `new`*xxx* and `free`*xxx* routines work in this way. `Free`*xxx* will cause a run time error if it is passed a `nil` pointer. `New`*xxx* will always initialize the `next`*xxx* field of the *xxx* record to `nil` so that the free list is cleanly isolated from any existing data structures. `Freecover` is a little more sophisticated. It frees all descendant `coverrecs` and `vlicomplexes` by calling itself recursively. One must make sure that all settings of `leaf` are correct before calling `freecover`.

6.2 Utility Routines (Section 3).

`Covers` and `intersects` are boolean functions which determine if one complex covers (or intersects) another complex. One complex covers another if every reference in the first complex is a superset of the corresponding reference in the second complex. Two complexes intersect if all of their references intersect.

`Unite` does a reference-wise union of two complexes. This is a generalization step since the resulting complex can cover more points in the event space than either complex did originally.

`Firstelement` and `lastelement` are functions which take a `refset` and find the first (lowest) and last (highest) element in the set. They don't abort if the set is empty, but they don't return any useful value either.

`Trim` implements the functional sort that has been used in many Michalski programs. An ordered set of cost functions with associated tolerances (relative and absolute) is provided. This function behaves identically to the cost function described in Larson's thesis. `Trim` is passed an `itemarray` which is an array of variant records called `items`. Each `item` can contain all types necessary (i.e. integers, `vl1cptrs`). The items are first sorted by cost function 1. Then any `items` which are within tolerance of the `quota item` are sorted by cost function 2, etc. The `quota item` is the item with the *n*th smallest cost where *n* is the quota of `items` desired. (After sorting, it is located at `item[`*n*`]`). If a tolerance is a whole number, then it is interpreted as an absolute tolerance. If it is a fraction, it is interpreted as a relative tolerance. Tolerances are entered to the program as percentages (i.e. 100 times their actual values).

There are print routines to print practically anything in the program. You cannot, however, print `vl1complexes` unless they correspond to the `l2symboltable`. This is because `printvl1c` uses the `l2symboltable`. This could be changed with a lot of editing.

6.3 Debugging.

As noted in the first few lines of the source program, various sets of debugging code can be turned on by doing appropriate global substitute commands in the editor and recompiling.

6.4 Modifying the Program for a Different Application.

The main constraint limiting the generality of the program is the `refsyms` approach to semantics. Any domain which can use that same approach (e.g. Letter Series Completion), can probably be solved using this program.

To move to a new application, layers 4 and 5 need to be removed and new front-ends need to be written. The procedures `level4examine` and `level4convert` would need to be rewritten. In the simplest case, the front end could accept `vlicomplexes` from the user, call level 3 to generalize them, and print out `vlicomplexes`. In such a case, `level4examine` would simply call `printvlic`. `Level4convert` would not be needed unless a CR and PE were desired.

The process of converting to a new domain is therefore not too difficult unless the new domain requires a smart, fancy front end like Eleusis did. Mostly, it is a big editing task.

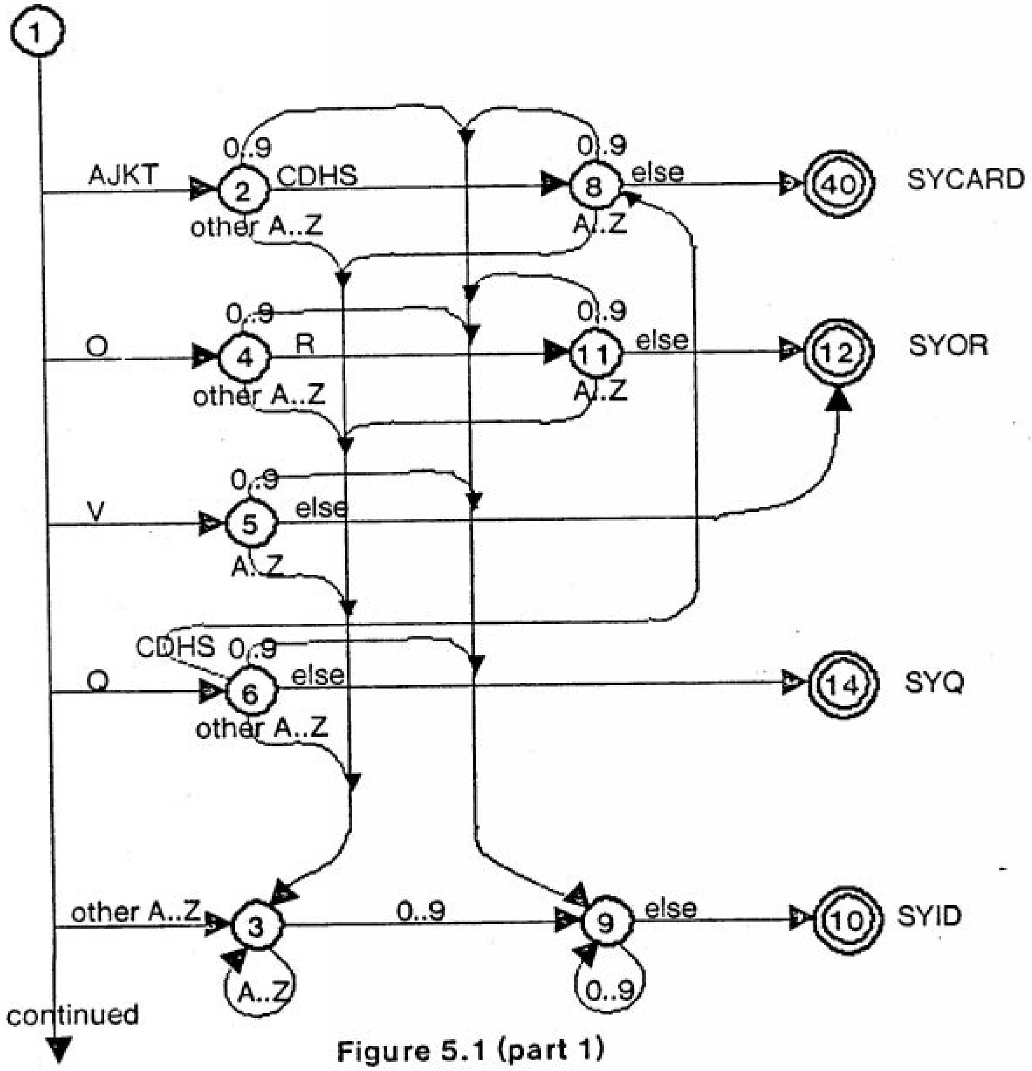


Figure 5.1 (part 1)

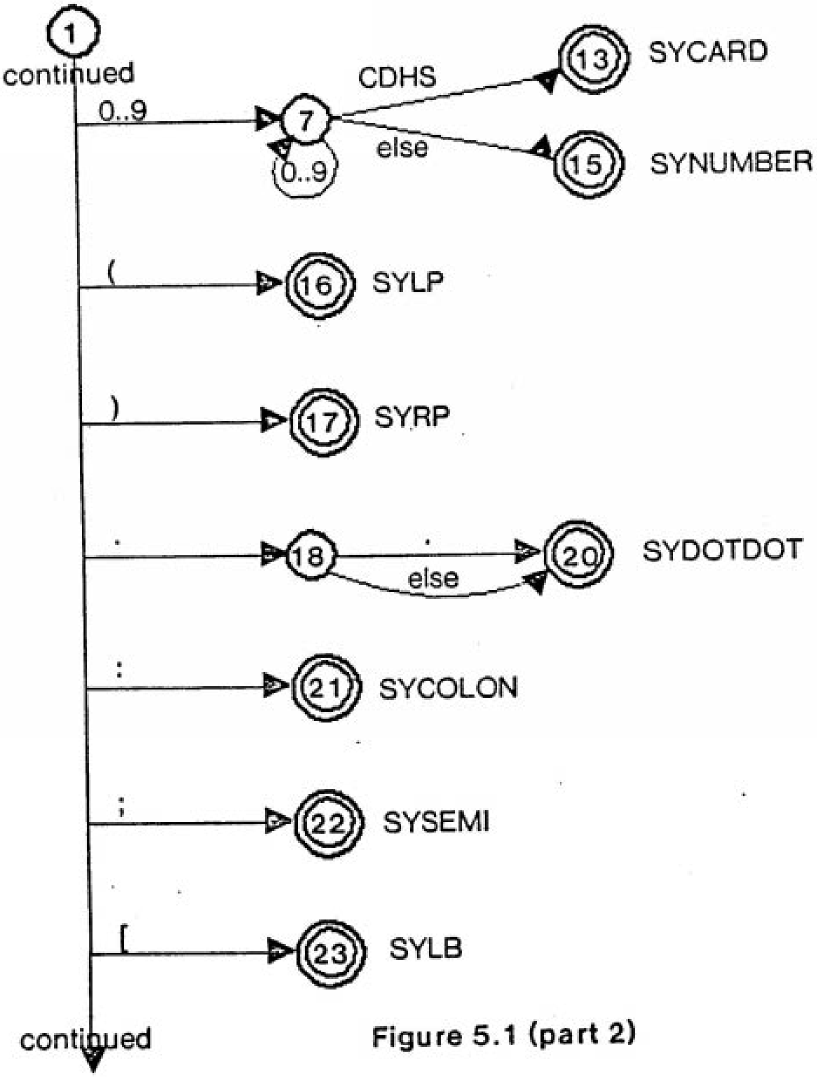


Figure 5.1 (part 2)

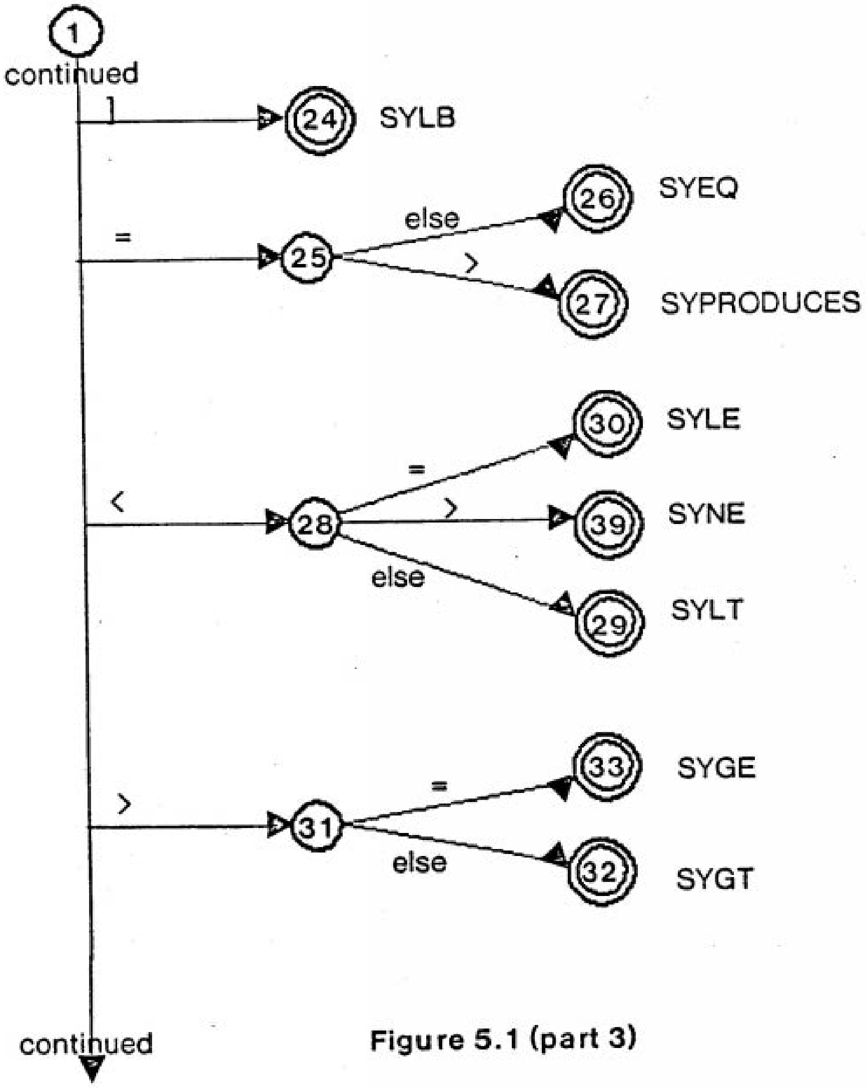


Figure 5.1 (part 3)

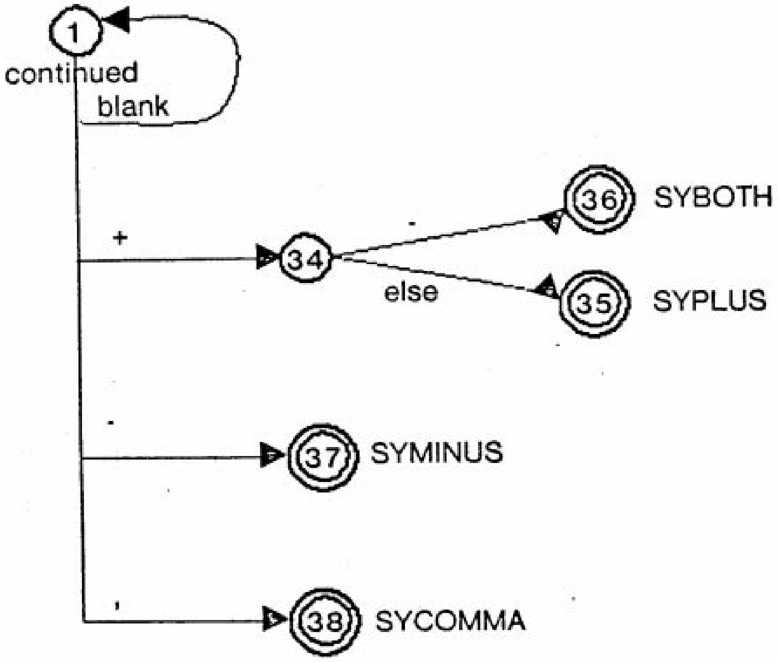


Figure 5.1 (part 4)