

FuzzyCLIPS 6.02A Architecture Manual

February 1995

R. Orchard

National Research Council
Institute for Information Technology
Knowledge Systems Laboratory

1.0 Overview

This report describes details of changes made at the National Research Council to implement a fuzzy expert system shell on top of CLIPS ([1], [2]). This extended version of CLIPS is called FuzzyCLIPS. The modifications made to CLIPS contain the capability of handling fuzzy concepts and reasoning. It enables domain experts to express rules using their own fuzzy terms. It allows any mix of fuzzy and normal terms, numeric-comparison logic controls, and uncertainties in the rules and facts. Fuzzy sets and relations deal with fuzziness in approximate reasoning, while certainty factors for rules and facts manipulate the uncertainty.

Unlike the *FuzzyCLIPS User's Guide*, this document will focus on modifications to structures used in standard CLIPS as well as new structures added to support the fuzzy extensions. It will provide a description suitable for the maintainers of FuzzyCLIPS and for those who wish to extend/modify the system. It assumes that the reader has some understanding of the standard CLIPS architecture.

Also note that the descriptions in this document will not replace reading the code to understand the way the extensions have been added and algorithms have been implemented. It will, however, serve the purpose of making the code much easier to understand.

2.0 Setup.h and constant.h File Changes

The file setup.h is used to record flags that can be set that determine the type of system on which the CLIPS system will be built (e.g. MAC_MPW, UNIX_V). It also has flags that determine the features of the system that will be included or left out when the system is built (e.g. OBJECT_SYSTEM). In FuzzyCLIPS several flags have been added to allow certain features of FuzzyCLIPS to be included or omitted. These flags are:

```

FUZZY_DEFTEMPLATES
CERTAINTY_FACTORS
EXTENDED_RUN_OPTIONS

```

They can be set to 0 to exclude the features they represent or to 1 to include them. FUZZY_DEFTEMPLATES and CERTAINTY_FACTORS are self explanatory and EXTENDED_RUN_OPTIONS is described in the User Guide.

The file constant.h has also been modified to add constant definitions for FuzzyCLIPS extensions.

(NOTE: If we extend FuzzyCLIPS to give fuzzy values FULL type status, like numerics, perhaps we need to define FUZZY_VALUE_TYPE_NAME
)

```

/*****/
/* FUZZY REASONING */
/* */
/* NOTE next 5 values must be less than */
/* PRIMITIVES_ARRAY_SIZE */
/* to fit in PrimitivesArray */
/* */
/* In standard CLIPS use constant 70 */
/* for size of PrimitivesArray */
/* */
/*****/

#define SCALL_PN_FUZZY_VALUE 69

#define SINGLETON_EXPRESSION 70
#define S_FUNCTION 71
#define Z_FUNCTION 72
#define PI_FUNCTION 73

#define PRIMITIVES_ARRAY_SIZE 75

```

```
/******  
/* FUZZY REASONING and CERTAINTY FACTORS */  
/* */  
/* Added at NRCC */  
/* */  
/******
```

```
#define PRIMARY_TERM      201  
#define MODIFIER          202  
#define FUZZY_VALUE      203
```

```
#define CRISP_LHS          219  
#define FUZZY_LHS         220
```

```
#define MAXMIN 1  
#define MAXPROD 2
```

```
#define FUZZY_TOLERANCE 0.00000001
```

These definitions are used in various places in the extensions.

3.0 Fuzzy Deftemplates (Fuzzy Variables or Linguistic Variables)

When defining a fuzzy deftemplate using the deftemplate construct, a special structure is created to hold the definition of that fuzzy deftemplate. These are also referred to as fuzzy variables or linguistic variables. The definition of a fuzzy variable includes its name, the universe of discourse for the variable, and the primary terms that can be used to describe the variable. A fuzzy deftemplate has the general format:

```
(deftemplate <name> [“<comments>”]
  <from> <to> [<unit>] ; universe of discourse
  (
    t1
    .
    ; list of primary terms
    .
    tn
  )
)
```

<name> is the identifier used for the fuzzy variable. The description of the universe of discourse consists of three elements. The <from> and <to> are floating point numbers that represent the begin and end of the interval which describes the domain of the fuzzy variable (the universe of discourse). The value of <from> must be less than the value for <to>. The <unit> is a **word** which represents the unit of measurement (optional). The t_i are specifications for the fuzzy terms (such as hot, cold, warm) used to describe the fuzzy variable. For example we might define the following fuzzy deftemplate:

```
(deftemplate temp ;linguistic variable ‘temp’
  0 100 C ;universe of discourse
  ( ; linguistic term definitions
    (low (PI 20 20)) ; term ‘low’
    (medium (PI 20 50)) ; term ‘medium’
    (high (PI 20 80)) ; term ‘high’
  )
)
```

When the deftemplate construct is parsed a deftemplate structure is created. It has the following data structures as defined in tmpltdef.h.

```
struct deftemplate
{
  struct constructHeader header;
  struct templateSlot *slotList;
  unsigned int implied : 1;
  unsigned int watch : 1;
  unsigned int inScope : 1;
  unsigned int numberOfSlots : 13;
  long busyCount;
```

```
    struct factPatternNode *patternNetwork;
#ifdef FUZZY_DEFTEMPLATES
    /* set to true if any fuzzy slots in deftemplate */
    unsigned int hasFuzzySlots : 1;
    /* will be a NULL ptr if not a fuzzy template */
    struct fuzzyLv *fuzzyTemplate;
#endif
};
```

Note that this has two extra fields defined that are not in the standard CLIPS structure. The field, fuzzyTemplate, points to another structure that holds the fuzzy template information. For non-fuzzy deftemplates (ie. those that do not define fuzzy variables) this will be a NULL pointer. The field hasFuzzySlots will be set to 1 if the template has any slots that can be fuzzy values. for fuzzy deftemplates that define fuzzy variables (and have exactly 1 slot which must hold a fuzzy value) this will be set to 1. If a template has no fuzzy slots this will be 0. The fuzzyLv structure is defined in fuzzylv.h as:

```
/* ***** */
/* LINGUISTIC VARIABLE STRUCTURE: */
/* ***** */
struct fuzzyLv
{
    /* Universe of Discourse */
    double from;
    double to;
    struct symbolHashNode *units;
    /* Primary Terms allowed */
    struct primary_term *primary_term_list;
};
```

The from and to floating point values hold the limits of the universe of discourse defined in the deftemplate. The units field is a pointer to a hash node in the symbol hash table where the actual string is stored. If the units specification is omitted then a NULL pointer is stored here.

Primary_term_list is a pointer to a list of primary terms defined in the deftemplate (there must be one or more terms).

Each primary term that is defined in the deftemplate is stored on a linked list with nodes that are defined as follows:

```
/* ***** */
/* PRIMARY_TERM STRUCTURE: */
/* ***** */
struct primary_term
{
```

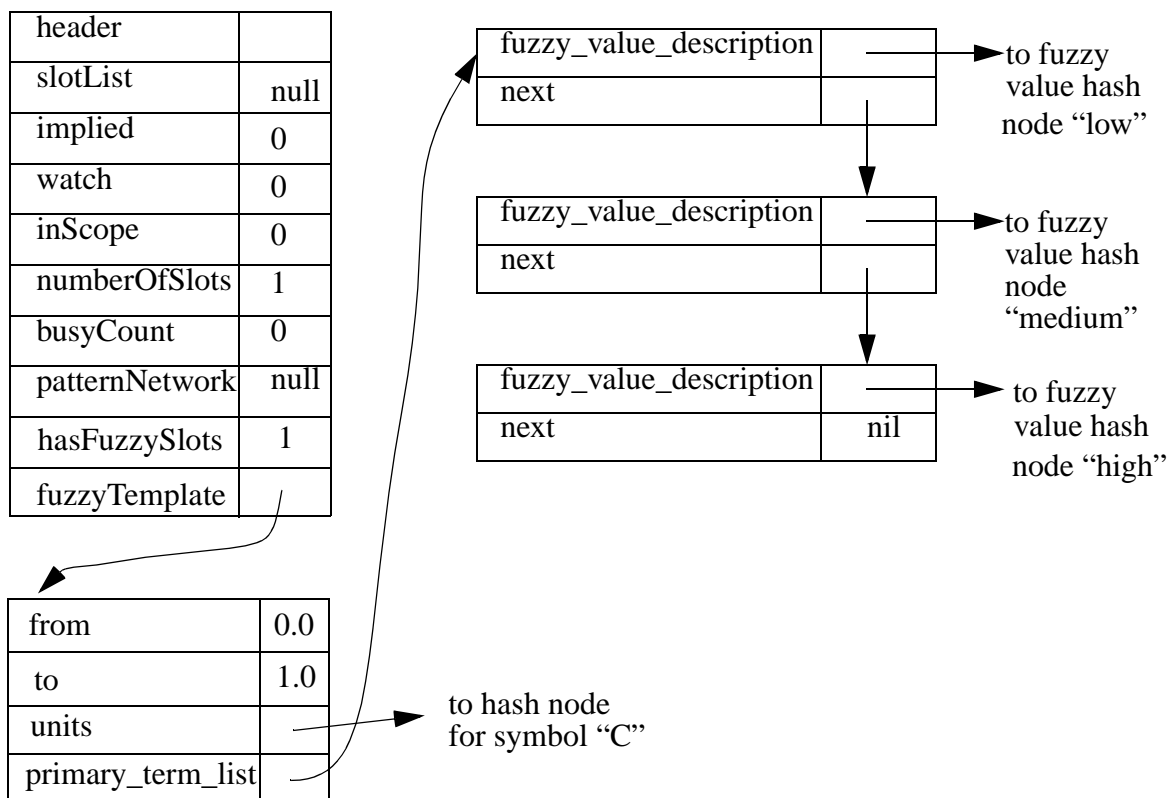
```

FUZZY_VALUE_HN *fuzzy_value_description;
struct primary_term *next;
};

```

The next field is a pointer to the next node in this list and the last node of the list has a NULL pointer here. The fuzzy_value_description field is a pointer to the fuzzy value hash table where the actual fuzzy value (fuzzy set) description is stored. The details of a fuzzy value description can be found in the next section.

Given the example above for deftemplate *temp*, the deftemplate and associated structures can be represented by the following diagram.



These structures are created during the compilation of a deftemplate that has a fuzzy template description. The function ParseDeftemplate in file tmpltpr.c is the starting point for parsing deftemplate constructs. After the name and comment fields are parsed, the modified function checks to see if the next token is a number (float or integer). If it is, then the function ParseFuzzyTemplate (implemented in file fuzzypsr.c) is called to handle the special case. If successful it returns a fuzzyLv structure, sets up a slot to add to the slot list (with slotName of "genericFuzzySlot"), and sets the numberOfSlots field to 1. The fuzzyTemplate field is set to point to the fuzzyLv structure returned from ParseFuzzyTemplate. When the deftemplate is 'installed' (InstallDeftemplate in tmpltpr.c), InstallFuzzyTem-

plate (in file fuzzypsr.c) is also called to install the required parts of the fuzzyLv (symbols for the units as well as the FuzzyValues associated with the primary terms).

NOTE: Slots in deftemplates that can hold Fuzzy Value types cannot hold any other types!

4.0 Fuzzy Value (Fuzzy Set) Structures

A fuzzy value is created to store the description of a membership function (fuzzy set) that is defined/used in a number of places:

- the primary terms in a fuzzy deftemplate specification
- a fuzzy pattern on the lhs of a rule
- an assert of a fuzzy fact
- in a deffacts specification for a fuzzy fact

The definition of the structure that holds fuzzy values is in fuzzyval.h and is shown below.

```

/*****
/* FUZZY_VALUE STRUCTURE:
/* name of the fuzzy value (linguistic expression)
/* maxn - size of x and y arrays
/* n - number of elements in x and y in use
/* x,y - the membership values in 2 arrays
/*
/* NOTE: at some time FuzzyValues should become atomic types. We
/* go most of the way by making them hashed etc.
/*
*****/
struct fuzzy_value
{
    struct deftemplate *whichDeftemplate; /* the template (fuzzy) */
    struct symbolHashNode *name; /* the fuzzy value linguistic
                                /* expression eg. "very cold" */

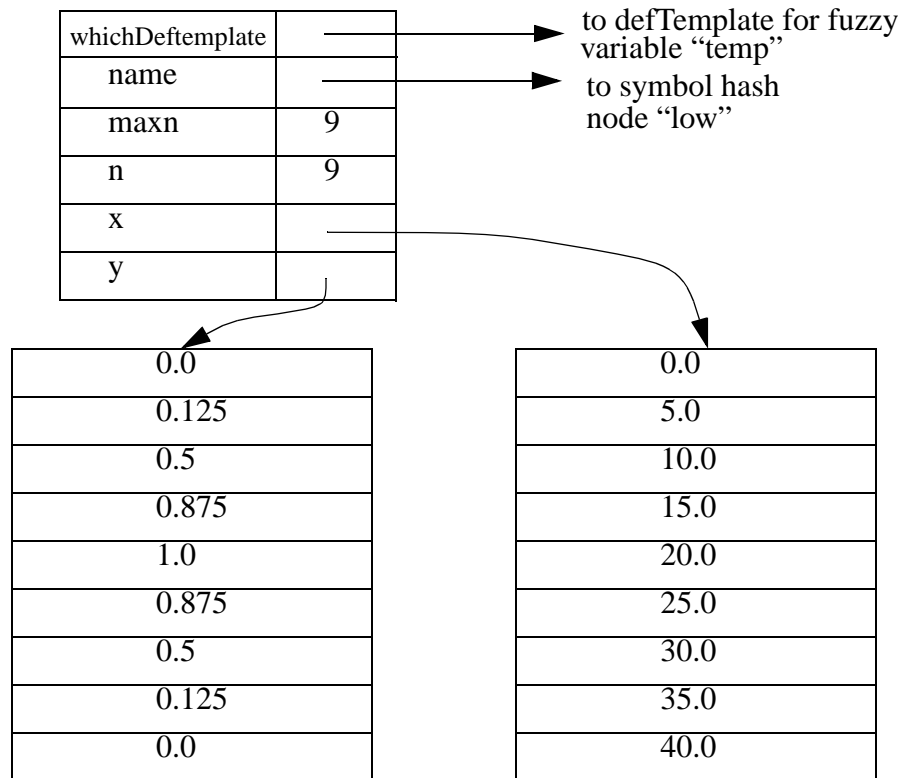
    int maxn;
    int n;
    double *x;
    double *y;
};

```

The variable whichDeftemplate is a pointer to the deftemplate that defines the Fuzzy Variable from which this fuzzy value was derived. The name variable is a pointer to an entry into the symbol hash table. It holds the name of the fuzzy value. This is the linguistic description of the fuzzy value. For primary terms this will be simple words such as 'hot', 'cold', or warm. For fuzzy pattern expressions or fuzzy facts this might be a more complicated expression such as 'very cold OR very hot'. In many cases where a fuzzy value (fuzzy set) is calculated by fuzzy inferencing, the exact linguistic expression to describe the fuzzy set is either not describable using the terms and modifiers or not easy to determine. In these cases the symbol '???' is used as the name of the fuzzy set.

When space is allocated to store the x and y values, the size of the allocated arrays (number of elements in the arrays) is stored in maxn. As elements are stored in the array the actual length of the arrays is placed in n.

The example from the fuzzy template section would produce three fuzzy values for the primary terms *low*, *medium* and *high*. The fuzzy value for low might have the following internal representation.



The fuzzy values are stored in a hash table in a manner similar to symbols, integers, and floats. There are a number of values and tables constructed that parallel these other hash tables for fuzzy values. Consider the following definitions extracted from symbol.h.

```
#define FUZZY_VALUE_HASH_SIZE 167

/*****
/* fuzzyValueHashNode STRUCTURE: */
*****/
struct fuzzyValueHashNode
{
    struct fuzzyValueHashNode *next;
    int count;
    int depth;
    unsigned int markedEphemeral : 1;
};
```

```

    unsigned int neededFuzzyValue : 1;
    unsigned int bucket : 14;
    struct fuzzy_value *contents;
};

typedef struct fuzzyValueHashNode FUZZY_VALUE_HN;

#define ValueToFuzzyValue(target) (((struct fuzzyValueHashNode *) (target))->contents)

#define IncrementFuzzyValueCount(theValue)
    (((FUZZY_VALUE_HN *) theValue)->count++)

LOCALE VOID      *AddFuzzyValue(struct fuzzy_value *);
LOCALE int       HashFuzzyValue(struct fuzzy_value *,int);
LOCALE VOID      DecrementFuzzyValueCount(struct fuzzyValueHashNode *);
LOCALE struct fuzzyValueHashNode  **GetFuzzyValueTable(void);
LOCALE VOID      SetFuzzyValueTable(struct fuzzyValueHashNode **);

```

The routines are added in symbol.c. Related to the creation and removal of entries in the fuzzy value hash table are routines *InstallFuzzyValue* and *DeinstallFuzzyValue* found in file fuzzypsr.c. These routines are used to do the incrementing and decrementing of the use counts in the hash table entries for the fuzzy values. They also will call routines to increment and decrement use counts in the symbol table entries for the names associated with the fuzzy values. The routines *InstallFuzzyTemplate* and *DeinstallFuzzyTemplate* are also found in the file fuzzypsr.c. These routines install and deinstall (increment and decrement use counts) the hash table entries associated with fuzzy deftemplates.

Since the hash table for fuzzy values is so similar to other hash tables we will not elaborate on the details of hash table construction and usage.

5.0 Changes to FACT Structure

In FuzzyCLIPS facts may be standard CLIPS facts or fuzzy facts. Fuzzy facts are of 2 types: fuzzy deftemplate facts have a single slot which holds a fuzzy value and a relation name which is the same as the fuzzy deftemplate definition; fuzzy facts are standard deftemplate facts that have 1 or more fuzzy slots defined. Facts can also have a certainty factor associated with them. The fact structure now has the following fields (see factm-ngr.h).

```
struct fact
{
    struct patternEntity factHeader;
    struct deftemplate *whichDeftemplate;
    VOID *list;
    long int factIndex;
    unsigned int depth : 15;
    unsigned int garbage : 1;
#ifdef CERTAINTY_FACTORS
    double factCF;
#endif
    struct fact *previousFact;
    struct fact *nextFact;
    struct multifield theProposition;
};
```

The only added field is factCF which allows the certainty factor to be stored with the fact details. This is simply a floating point value in the range 0.0 to 1.0.

The field *theProposition* is a multifield structure as shown below (see multifld.h).

```
struct multifield
{
    unsigned busyCount;
    short depth;
    unsigned short multifieldLength;
    struct multifield *next;
    /* struct field HUGE_ADDR theFields[1]; */
    struct field theFields[1];
};

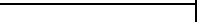


struct field
{
    short int type;
    VOID *value;
};
```

In the case of ordered (non-deftemplate facts) multifieldLength is always 1 and there is one entry in theFields array of type multifield to store the fields of the fact. If the fact is a

template fact then theFields array will be large enough to hold each field of the template and each field may be of type Integer, Float, Multifield, Fuzzy Value, etc. See factcom.c for the routine AssertCommand to see how facts are constructed.

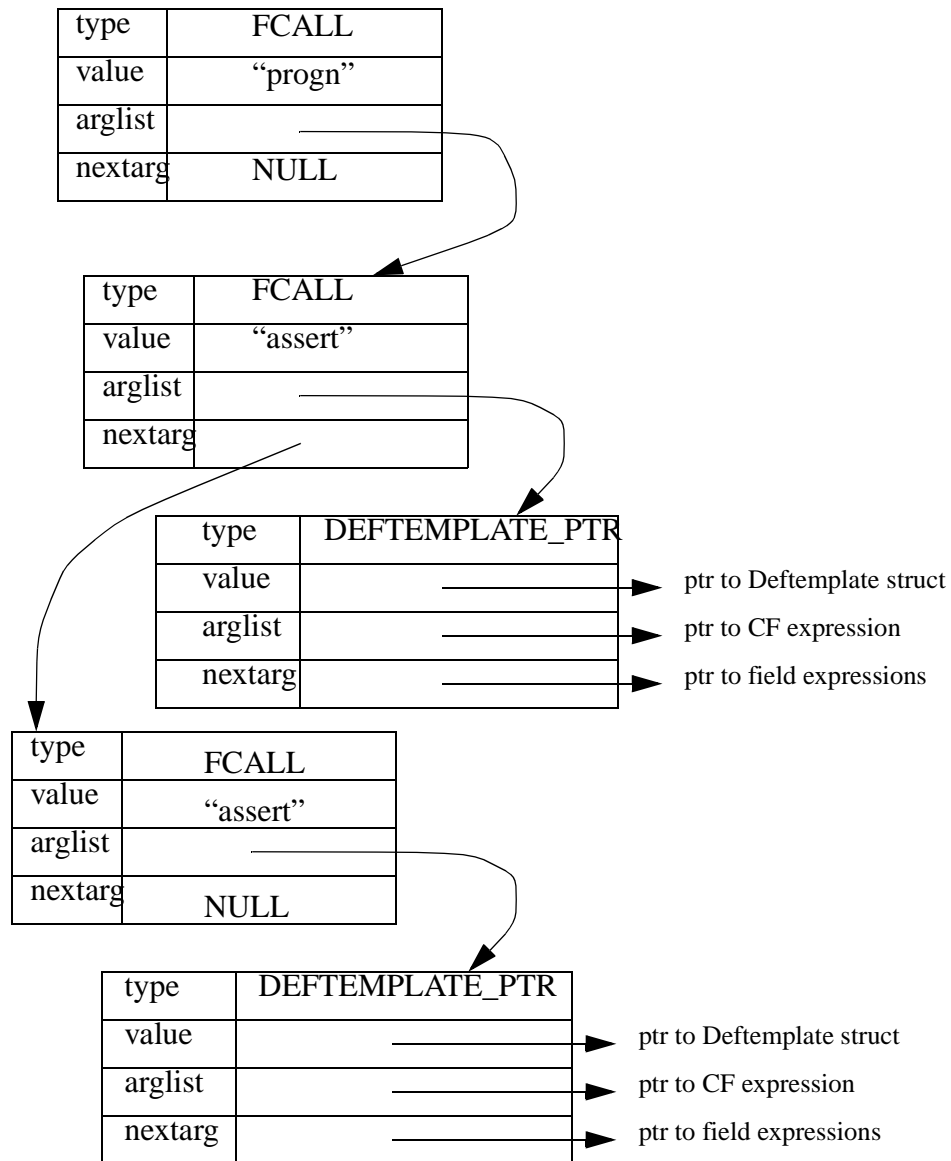
6.0 Assert and Deftemplate Structures

Facts (fuzzy deftemplate, with fuzzy slots or standard) are created by the assert command and assert-string function or in deffacts constructs. In the case of the deffacts construct and the assert command, when these are parsed a call to BuildRHSAAssert (in file factrhs.c) is made. This function in turn calls GetRHSPattern for each fact definition included in the deffacts construct or assert command. The assert-string function calls GetRHSPattern only since it can include a single fact in the string argument. Each call to GetRHSPattern returns an expression of type DEFTEMPLATE_PTR (shown below).

type	DEFTEMPLATE_PTR
value	 ptr to Deftemplate struct
arglist	 ptr to CF expression
nextarg	 ptr to field expressions

The value is a pointer to the deftemplate structure for the fact. Normally arglist is NULL for standard CLIPS facts. For FuzzyCLIPS this is a pointer to an expression that when evaluated supplies the CF or certainty factor of the fact (if NULL a CF of 1.0 is assumed). Nextarg points to expressions for the slots of the fact. In the case of a fuzzy fact (a fact with fuzzy slots or a fuzzy deftemplate fact), the expressions for the fuzzy slots will be expressions that when evaluated return a Fuzzy Value. This is detailed later in this section. When a fuzzy slot is being parsed the routine GetRHSPattern will call ParseAssertFuzzy-Fact (found in fuzzyrhs.c) for a fuzzy deftemplate or ParseAssertTemplate for fuzzy slots in a deftemplate to handle the parsing of the slot.

If the call to GetRHSPattern was made from BuildRHSAAssert (as is the case when parsing an assert command or a deffacts construct) then the DEFTEMPLATE_PTR expression returned by GetRHSPattern is wrapped in an FCALL expression for an assert, with the DEFTEMPLATE_PTR expression as the single argument for the assert (linked via the arglist slot). If there are several facts (in the assert or deffacts) then these FCALLs to assert are wrapped in a FCALL to progn expression. When a reset is done the expression linked to the deffacts construct will be executed and each call to the assert function will evaluate use the DEFTEMPLATE_PTR expression to create a fact. Similarly the expression that resulted from parsing the assert command when executed at runtime will cause facts to be created and asserted. Below we show the structures created for an assert with 2 facts being asserted. Unlike for the assert command and deffacts construct, the call to GetRHSPattern from the assert-string function is made at runtime. The returned DEFTEMPLATE_PTR expression is used then to make a fact and assert it.



As mentioned earlier the 'ptr to field expressions' shown in the example above points to expressions for the slots of the deftemplate. If the slot is a fuzzy slot the type of expression pointed to depends on the nature of the fuzzy specification. If a fuzzy deftemplate fact, for example, is specified using a linguistic expression such as

(temp very cold)

or a singleton expression with all constants such as

(temp (0 1) (5 .8) (10 .2) (15 0))

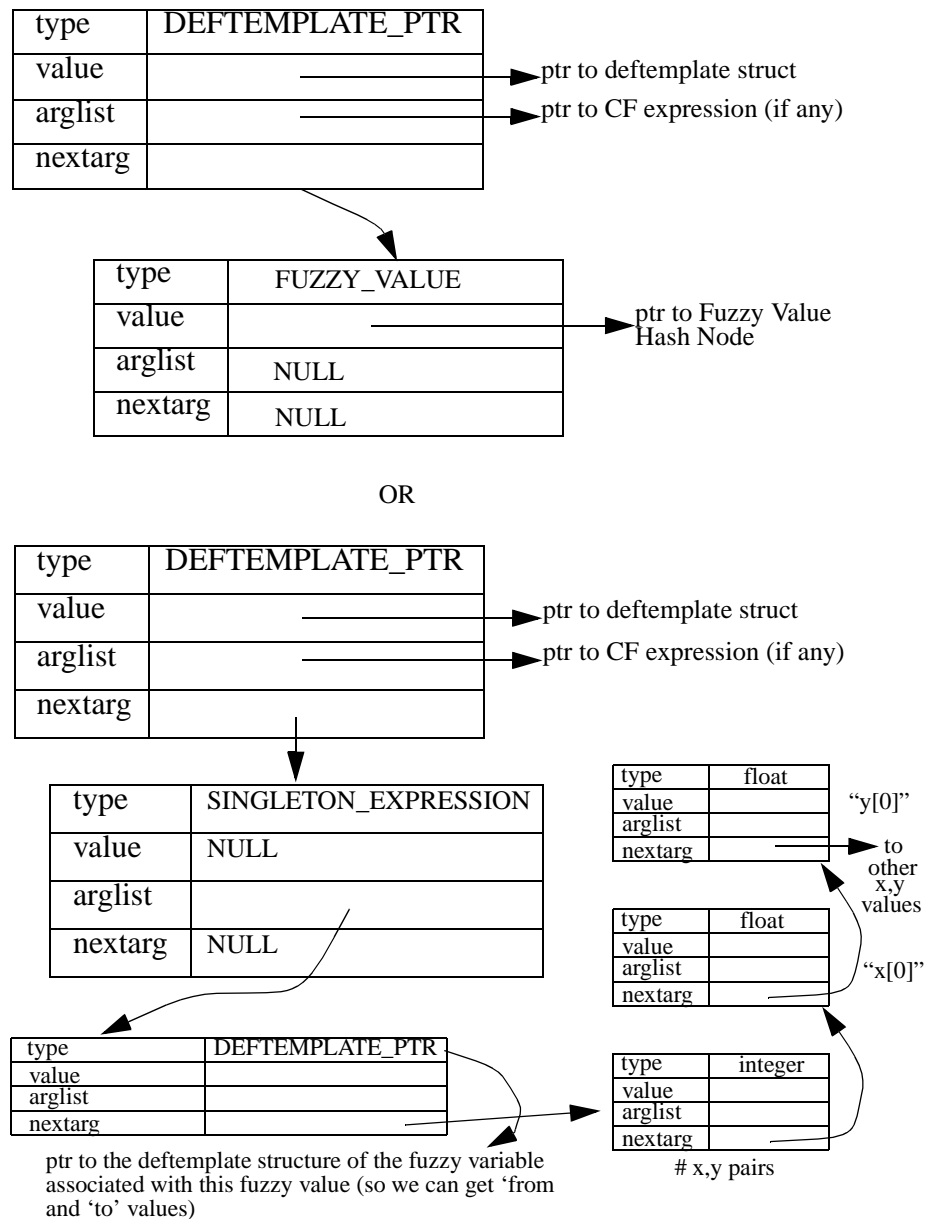
or a standard function (PI, S, Z) with all constants such as

```
(temp (Z 0 10))
```

then the expression pointed to will be of type FUZZY_VALUE. If the fuzzy fact is specified using expression or variables such as

```
(temp (Z (+ ?x1 ?x2) 0))
(temp (0 ?*g*) (10 (/ ?*g* 2)))
```

then the expression pointed to will be of type SINGLETON_EXPRESSION, PI_EXPRESSION, Z_EXPRESSION, or S_EXPRESSION. The format of these structures is detailed below.



In the second example in the diagram above, the DEFTEMPLATE_PTR node points to a SINGLETON_EXPRESSION. The first 2 arguments (pointed to by the arglist field of the SINGLETON_EXPRESSION node) are: a pointer to the deftemplate structure associated with this fuzzy value so we can access the 'from' and 'to' values as required (to check the range of the fuzzy set x,y pairs, etc.); the count of the number of (x,y) pairs in the singleton list (a constant integer value). These 2 arguments are followed by pairs of x and y values. At least one of the x or y values must be a non-constant expression or a Fuzzy Value would be created and returned. The structure for a standard function specification is similar to that of a singleton specification except that there are only 2 arguments after the *deftemplate_ptr* argument. These are the 2 arguments required for the Z, PI or S functions. Also the type of the node linked to from the DEFTEMPLATE_PTR node would be either Z_EXPRESSION, S_EXPRESSION, or PI_EXPRESSION as required rather than the type SINGLETON_EXPRESSION as shown here.

7.0 Changes to Rule and Agenda (Activation) Structures

The rule structure has had a number of fields added to accommodate fuzzy reasoning and certainty factors. The new structure found in ruledef.h is replicated below.

```
/* These structs are added at the end of rule structs
   that have patterns that have FUZZY slot references.
   They record for each fuzzy slot the pattern number
   in the rule and the slot number within the pattern
   as well as the ptr to that fuzzy values hash node.
```

The patterns and slots are indexed from 0 (i.e. the 1st pattern has patternNum = 0)

```
*/
struct fzSlotLocator
{
    unsigned int patternNum;
    unsigned int slotNum : 13;
    FUZZY_VALUE_HN *fvhnPtr;
};
```

```
struct defrule
{
    struct constructHeader header;
    int salience;
    int localVarCnt;
    unsigned int complexity : 11;
    unsigned int afterBreakpoint : 1;
    unsigned int watchActivation : 1;
    unsigned int watchFiring : 1;
    unsigned int autoFocus : 1;
    unsigned int executing : 1;
#ifdef DYNAMIC_SALIENCE
    struct expr *dynamicSalience;
#endif
    struct expr *actions;
#ifdef LOGICAL_DEPENDENCIES
    struct joinNode *logicalJoin;
#endif
    struct joinNode *lastJoin;
    struct defrule *disjunct;
#ifdef CERTAINTY_FACTORS
    double CF;
    struct expr *dynamicCF;
#endif
#ifdef FUZZY_DEFTEMPLATES
```

```

double    min_of_maxmins;
unsigned int lhsRuleType;
unsigned int numberOfFuzzySlots;
struct fzSlotLocator HUGE_ADDR *pattern_fv_arrayPtr;
#endif
};

```

For certainty factors we add two fields. The field CF holds the value of the certainty factor for the rule. This is specified in the rule's declare section:

```

(defrule A-rule-with-CF
  (declare (CF 0.8))
  ...
)

```

The field dynamicCF is a pointer to an expression that will be evaluated dynamically to determine the certainty factor for the rule. It is analogous to the dynamicSaliency and operates in a similar fashion. There are two functions, set-CF-evaluation and get-CF-evaluation, that are used to change and access a value that determines when the CF for the rule is evaluated. These functions are similar to set-saliency-evaluation and get-saliency-evaluation. An example of a rule that has a dynamic saliency specification follows:

```

(defrule A-rule-with-dynamic-CF
  (declare (CF (* 2.0 ?*g*)))
  ...
)

```

In this case the rule's certainty factor could be evaluated when the rule is define or when it is activated and placed on the agenda.

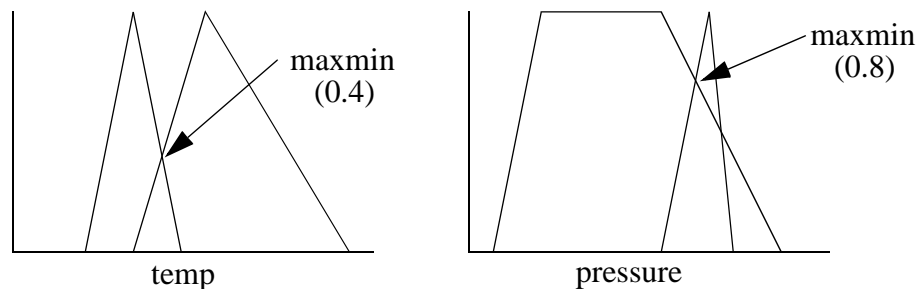
There are four fields added to support fuzzy reasoning. The field min_of_maxmins is used to hold the value of the minimum of maxmins for FUZZY-FUZZY type rules. A maxmin is in essence the maximum value in the intersection between the value (fuzzy set) of a fuzzy variable and the value (fuzzy set) for a fuzzy pattern for that variable on the LHS of a rule. Then min_of_maxmin is the minimum of all of the maxmins for the rule. For example, given the rule:

```

(defrule A-rule
  (temp low)
  (pressure high)
=>
  (printout t "A-rule fired" crlf)
)

```

A 'temp' fact's fuzzy set will intersect with the fuzzy set defined by 'temp low' and a 'pressure' fact's fuzzy set will intersect with the fuzzy set defined by 'pressure high'. This will provide two maximum values for the intersections. The minimum of these two values is the `min_of_maxmins`.



$$\text{min_of_maxmins} = \min(0.4, 0.8) = 0.4$$

When a fact (fuzzy) is asserted during the execution of the RHS of a rule this value is used to calculate the fuzzy consequence. The routine `computeFuzzyConsequence` (`fuzzyutl.c`) will call `computeMinOfMaxmins` (`fuzzyutl.c`) to calculate the actual `min_of_maxmins` when a fuzzy fact is asserted from a rule and the rule's LHS is FUZZY. The type of the rule's LHS is determined (if any fuzzy patterns then type is `FUZZY_LHS`) at the time a rule is compiled and is stored in the second new slot, `lhsRuleType`. When the rule is chosen to execute it's `min_of_maxmins` value is set to -1.0 to indicate that it has not yet been calculated. Once it has been calculated it will take on a value between 0 and 1. Thus it only gets calculated once per rule execution and not for every fuzzy fact that is asserted during the rule's execution.

The number of patterns in the rule that refer to templates with fuzzy slots is stored in the field `numberOfFuzzySlots`. If there are none then this will be zero. This value determines the number of entries in the array of Fuzzy Value slot locator structures that is pointed to by the `pattern_fv_arrayPtr` field. Entries in this array hold the pattern number and the slot number within the pattern of all fuzzy slots and a pointer to a fuzzy value hash node of the fuzzy value for the slot in the pattern. These locators are set when the rule is being compiled (routine `ProcessRuleLHS` in file `rulepsr.c`). The routine `computeMinOfMaxmins` (`fuzzyutl.c`) uses these pattern fuzzy values to perform the required `min_of_maxmins` calculation. When a fuzzy consequent is to be asserted the value of `max_of_maxmins` is used to 'clip' the consequent fuzzy set (the so-called max-min inference technique). This is performed in routine `computeFuzzyConsequence` in file `fuzzyutl.c`.

[
NOTE: the inference technique can be either max-min or max-dot (max-product). In routine `computeFuzzyConsequence` in file `fuzzyutl.c` the lines

```
conclusion = horizontal_intersection(consequence, ExecutingRule->min_of_maxmins);
```

and

```
conclusion = max_prod_scale(consequence, ExecutingRule->min_of_maxmins);
```

are used to do the correct form of inferencing.

]

The activation record (see below and file agenda.h) which is placed on the agenda when a rule's pattern matching is completed for a set of patterns must also be modified to reflect the requirements of fuzzy reasoning and certainty factors. All are actually related to certainty factors.

```
struct activation
{
    struct defrule *theRule;
    struct partialMatch *basis;
    int salience;
    unsigned long int timetag;
#ifdef CONFLICT_RESOLUTION_STRATEGIES
    struct partialMatch *sortedBasis;
    int randomID;
#endif
#ifdef CERTAINTY_FACTORS
    double CF;
    double StdConcludingCF;
#endif
#ifdef FUZZY_DEFTEMPLATES
    double FuzzyCrispConcludingCF;
#endif
#endif
    struct activation *prev;
    struct activation *next;
};
```

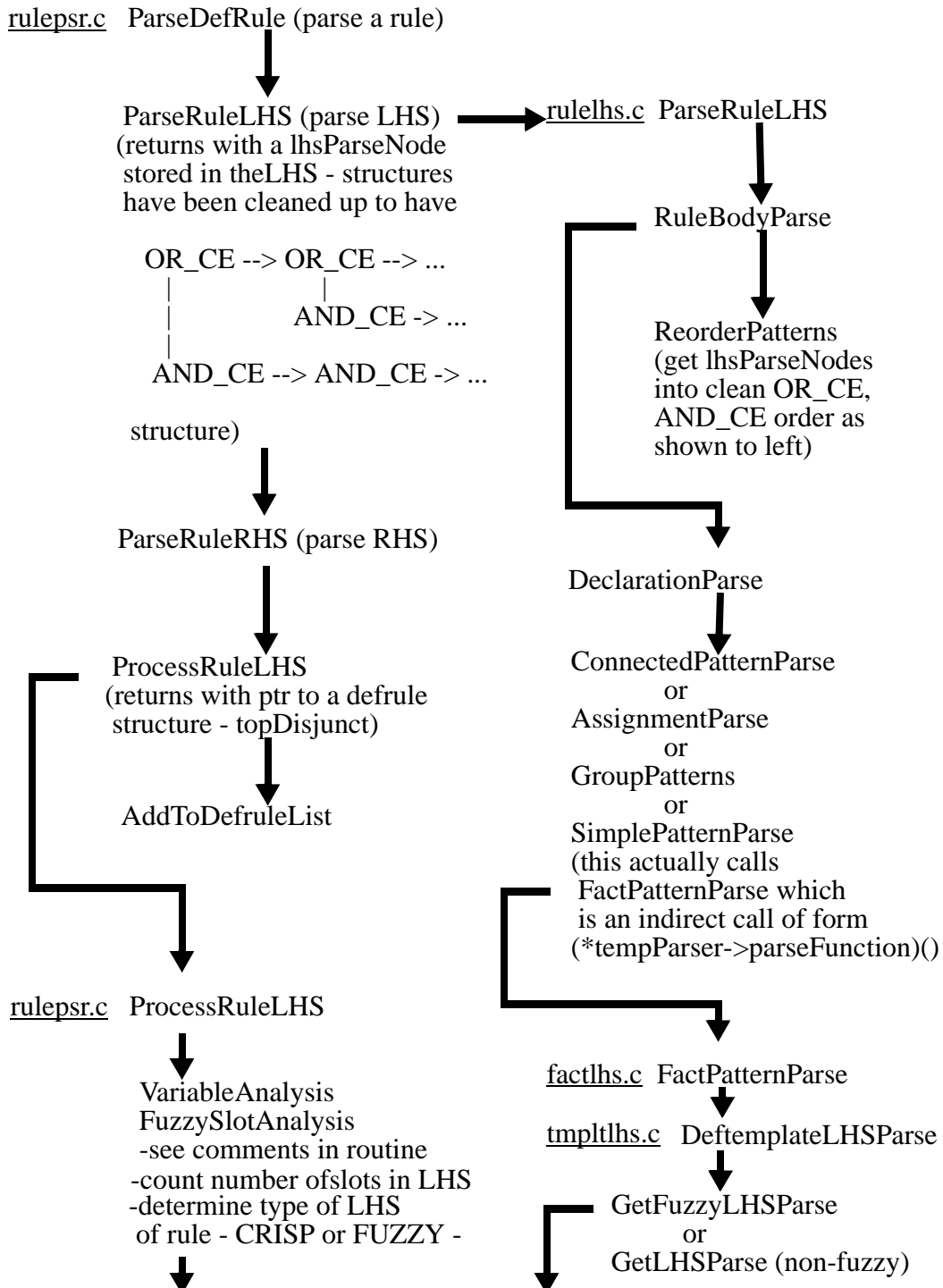
When a rule is added to the agenda an activation record is created. The rule's CF is obtained (calculated if dynamic CF and CF-evaluation is when-activated) and stored in the activations field CF. At this time both the StdConcludingCF and FuzzyCrispConcludingCF fields are set to -1.0 to indicate that they have not yet been calculated.

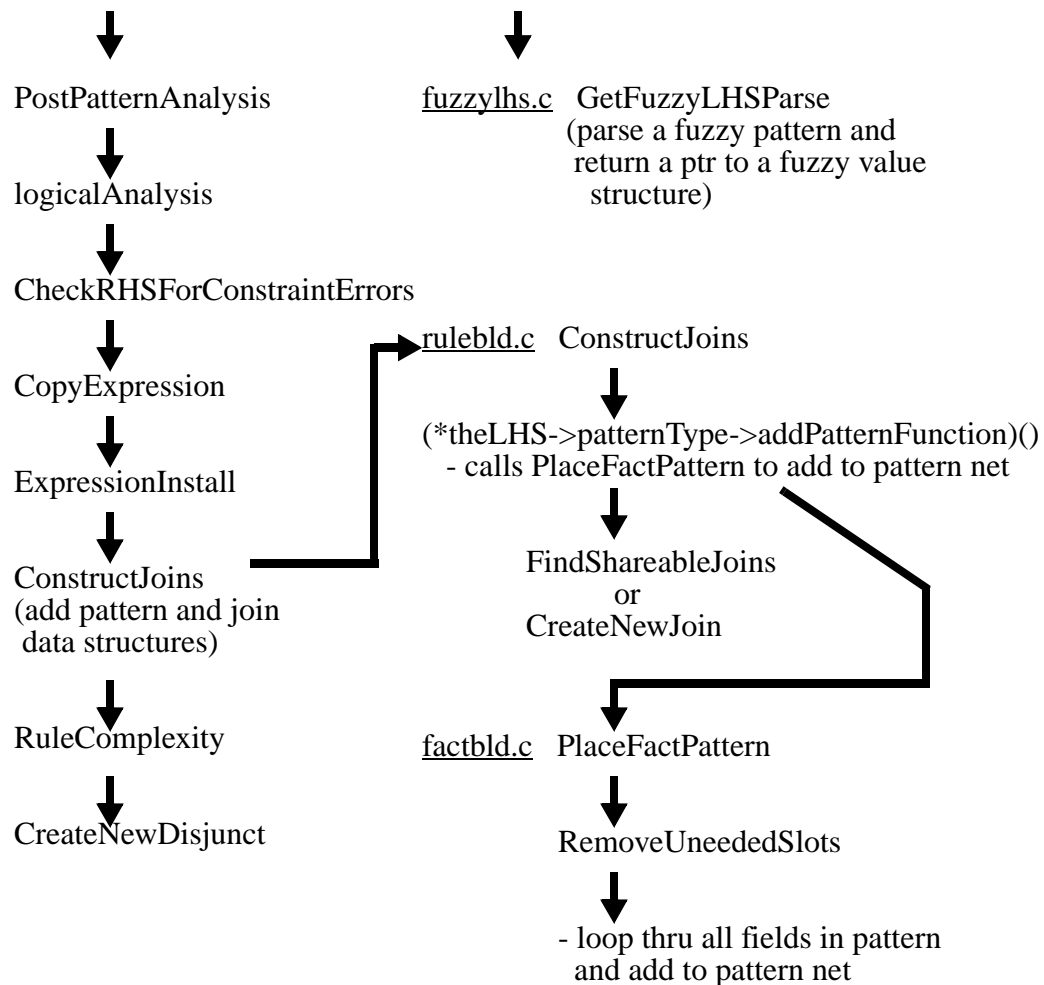
When the rule is activated (fired) the StdConcludingCF will be calculated (computeStdConclCF in cfdef.c) in the routine Run (engine.c) and compared against the Threshold_CF to determine if the rule should fire or not. StdConcludingCF is calculated based on the rule's certainty factor (in CF field) and the minimum of the CF's for all facts that matched patterns of the rule's activation ($CF * \min(\text{fact1}, \dots, \text{factn})$). The StdConcludingCF is used to assign certainty factors to all fuzzy facts that are asserted and to crisp facts if the LHS of the rule is CRISP. If the LHS of the rule is FUZZY then the FuzzyCrispConcludingCF

value will be calculated using `computeFuzzyCrispConclCF` (in file `cfdef.c`). When facts are being asserted, the routine `changeCFofNewFact` (defined in `cfdef.c` and called from `Assert` in `factmngr.c`) or `changeCFofExistingFact` (defined in file `cfdef.c` and called from `FactExists` in file `facthsh.c`) are called.

8.0 Pattern Matching of Fuzzy Facts

Various structures are created during the processing of the LHS patterns of a rule. It is instructive to map out the flow through the files and routines that occurs. This will be done in the following diagrams and after that we will present the detailed structures created at various points of this processing. First the flow through processing of the LHS of a rule.



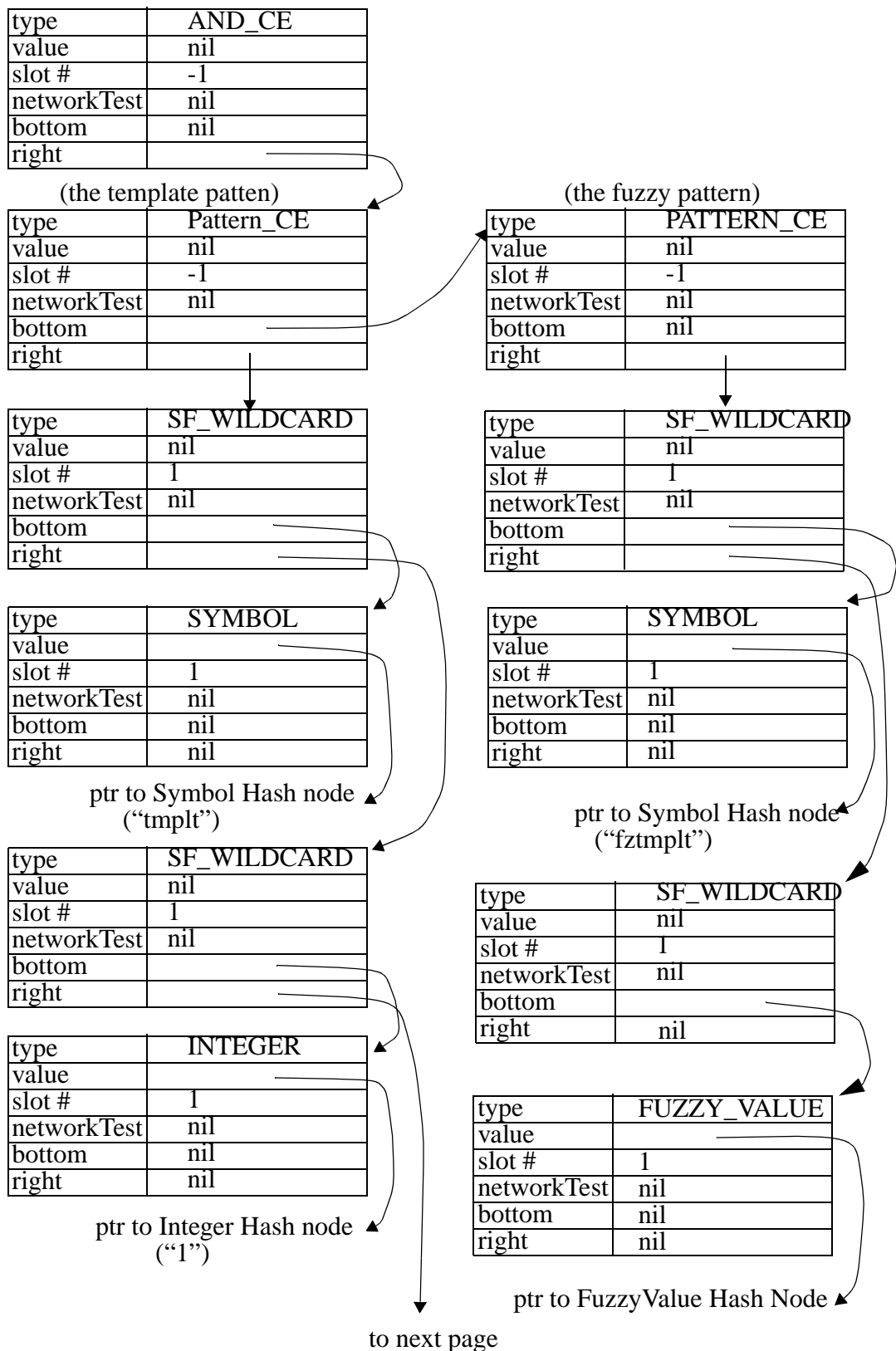


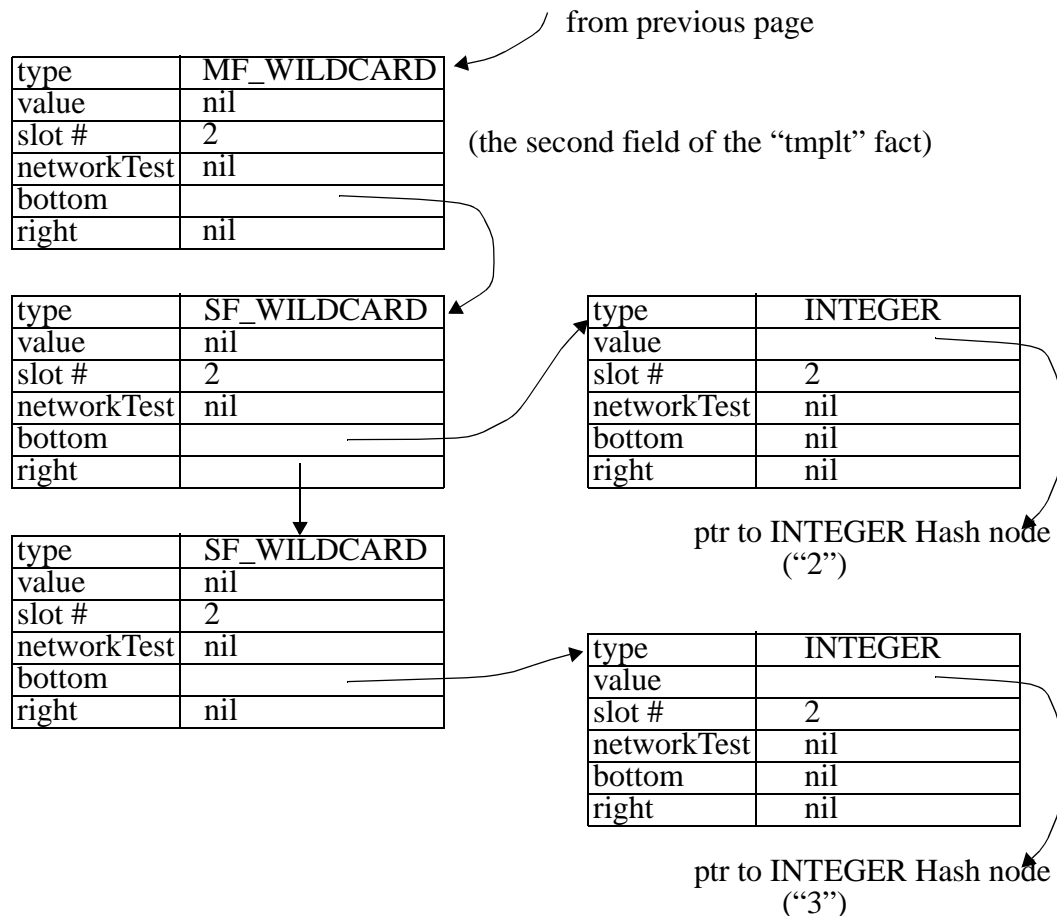
Consider now a simple set of patterns in a rule and we will follow the creation of structures as we go through this path shown above, highlighting things that are relevant to fuzzy patterns.

```

(defrule test
  (tmplt (a 1) (b 2 3))
  (fztmlt cold)
=>
)
  
```

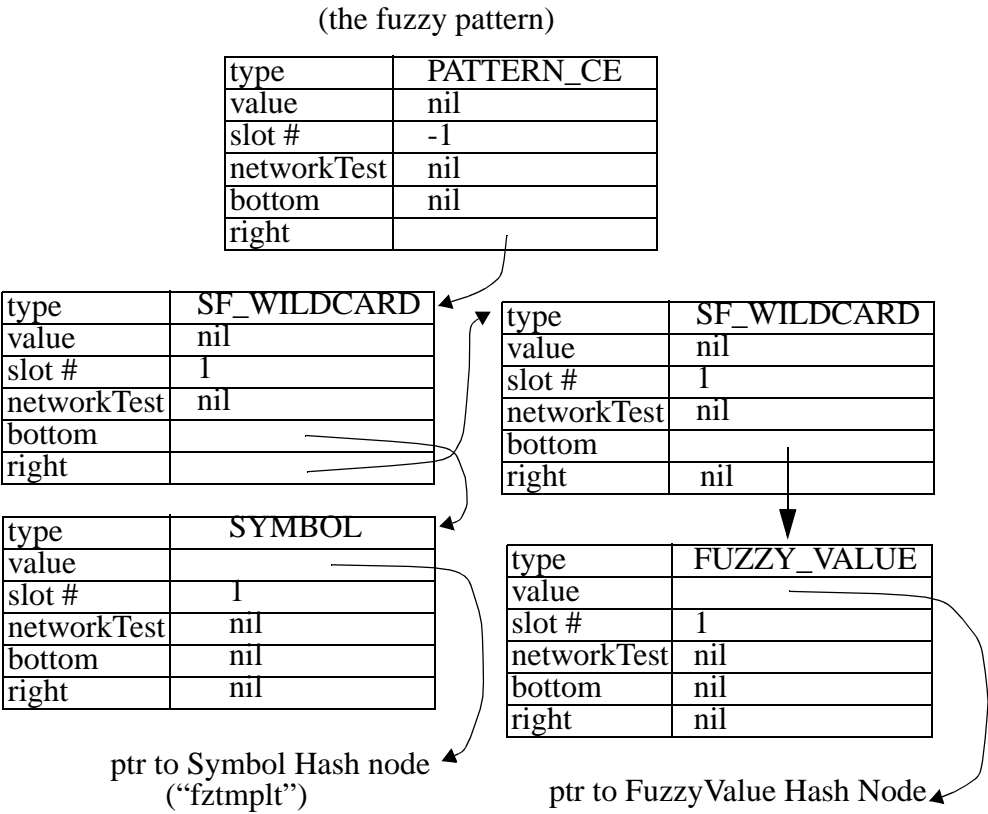
The first pattern is a template pattern with the template name **tmplt**. The second pattern is a fuzzy template pattern with the template name **fztmlt**. Slot **a** of **tmplt** is a single field slot and slot **b** is a multifield slot. After the routine **ParseRuleLHS** has completed in the routine **ParseDefRule**, the variable **theLHS** will point to the top of a set of **lhsParseNodes** that holds the structure of the parsed patterns and looks like the following (not all fields of **lhsParseNodes** are shown):





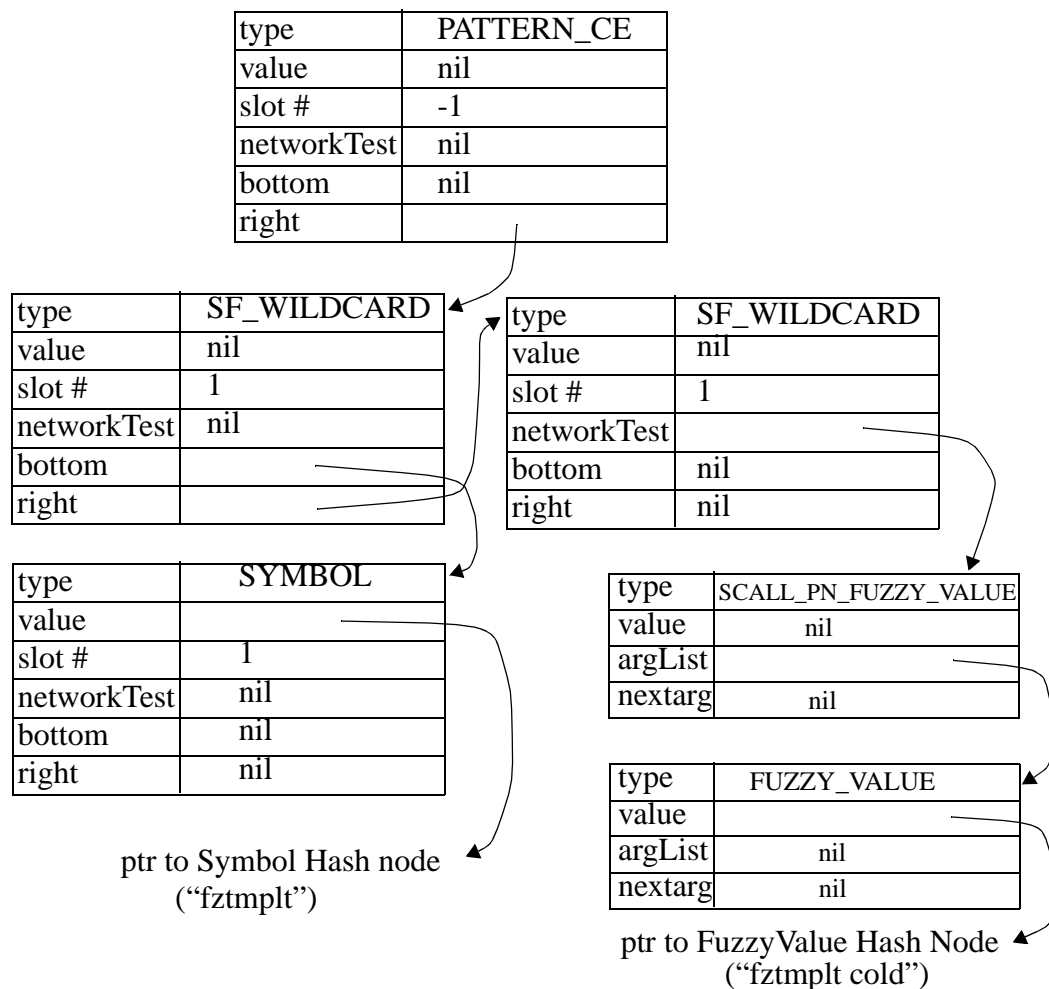
The next significant event (as far as fuzzy processing is concerned) is the call to ProcessRuleLHS. Within this routine the VariableAnalysis routine does a number of things including adding some networkTest nodes to the lhsParseNode structure that has been created. In this example it will add a pointer in the networkTest slot of all SF_WILDCARD lhsParseNodes. The pointer will point to an expression of type SCALL_PN_CONSTANT2 or SCALL_PN_CONSTANT4. This is done by standard CLIPS processing. The point here is not to rediscover what CLIPS does normally but to set up for what FuzzyCLIPS does with the lhsParseNodes of type FUZZY_VALUE. It does something quite similar to adding the networkTests for constants in order to reduce the time to do the compare of the FuzzyValue with the Fuzzy Pattern. Within ProcessRuleRHS the routine FuzzySlotAnalysis is also called (file analysis.c). This routine will perform some fuzzy checking on the types of patterns/slots and returns a count of the number of fuzzy slots in the patterns (0 if no fuzzy slots). Then the type of the LHS of the rule - FUZZY_LHS or CRISP_LHS - is set. In our example after executing the routine VariableAnalysis LHSRuleType is FUZZY_LHS and numPatterns is set to 2. It also replaces the SF_WILDCARD node's networkTest entry with a pointer to an SCALL_PN_FUZZY_VALUE node that is linked to the FUZZY_VALUE node (removing it from the 'bottom' pointer in the SF_WILDCARD node).

The effect of this change is as follows. Where previously in the lhsParseNodes structures shown above we had:



we now have:

(the fuzzy pattern)



In the routine PlaceFactPattern a new factPatternNode is added to the pattern network when necessary and the networkTest is added to that node. This is done by a call to the routine CreateNewPatternNode (also in factbld.c). At runtime when a fuzzy fact is being matched against the fuzzy pattern the routine FactFuzzyValuePNFunction (file factrete.c) will be called from EvaluatePatternExpression (file factmch.c) in a manner similar to that used for pattern net constant functions (e.g. FactConstantPNFunction4 for SCALL_PN_CONSTANT4).

9.0 Performing a Binary Save/Load (BSAVE/BLOAD)

There are no new modules added to FuzzyCLIPS to support the Binary Save and Load functions. Instead existing code throughout CLIPS has been modified as necessary to create correct binary files with the fuzzy extensions included. Files involved with changes are:

rulebin.h	rulebin.c
symlbin.h	symlbin.c
tmpltbin.h	tmpltbin.c
exprnbin.c	
bload.c	
bsave.c	

The **saving and restoring of defrules** as binary structures is handled for the most part in rulebin.c. A structure for saving rules is defined in rulebin.h (modified for Fuzzy extensions) and is shown below.

```
/* These structs are added at the end of rule structs
   that have patterns that have FUZZY slot references.
   They record for each fuzzy slot the pattern number
   in the rule and the slot number within the pattern
   as well as the ptr to that fuzzy values hash node.
*/
```

```
struct bsaveFzSlotLocator
{
    unsigned int patternNum;
    unsigned int slotNum : 13;
    long    fvhnPtr;
};

struct bsaveDefrule
{
    struct bsaveConstructHeader header;
    int salience;
    int localVarCnt;
    unsigned int complexity    : 12;
    unsigned int autoFocus    : 1;
    long dynamicSalience;
    long actions;
    long logicalJoin;
    long lastJoin;
    long disjunct;
#ifdef CERTAINTY_FACTORS
    double CF;
    long dynamicCF;
#endif
};
```

```
#endif
#if FUZZY_DEFTEMPLATES
    double    min_of_maxmins;
    unsigned int lhsRuleType;
    unsigned int numberOfFuzzySlots;
    long      pattern_fv_arrayPtr;
#endif
};
```

This is a quite straightforward extension and the routines in rulebin are modified accordingly to save the extra information. Note that dynamicCF is treated much like dynamicSaliency. However, the array pointed to by pattern_fv_arrayPtr in a fuzzy defrule poses more of a problem. To handle this we create an extra variable, array (ptr), and routine.

NumberOfPatternFuzzyValues - is a count of the total number of pattern fuzzy slots pointed to by all pattern_fv_arrayPtr fields in all defrules. This is just a sum of all of the numberOfFuzzySlots fields in all rules.

PatternFuzzyValueArray - this is an array to hold all of the fzSlotLocator structures found in all of the rules.

UpdatePatternFuzzyValues - This routine is used to set the pointers in the PatternFuzzyValueArray correctly as the binary saved file is loaded.

When a bsave is done all binary defrule structures are written out. Then all of the fzSlotLocators pointed to by pattern_fv_arrayPtr are written out. The long values written out in the fvhnPtr field of bsaveFzSlotLocators are not pointers to the Fuzzy Value hash node. Instead they are values which will be indices into FuzzyValueArray. This array will hold the pointers to the hash nodes when the bload is done. This is very similar to the way symbol hash node pointers are treated.

In effect when the FUZZY_VALUES (or SYMBOLS in the symbol hash table) marked as 'needed' are written out they are written in the order that the table is traversed. The 'bucket' slot for each node in the table is modified temporarily during a bsave to reflect this order (i.e. the slot will contain an integer index that reflects the order that the nodes were written to the binary file). Then when any reference to a hash node is written to the binary file, this value in the 'bucket' slot is written. When the binary file is read, the FuzzyValues (SYMBOLS) are read in the order they were written. The FuzzyValue (Symbol) is added to the hash table and a pointer to the real hash node is placed into the FuzzyValueArray (SymbolArray). The details of writing out FuzzyValue hash tables (and Symbol hash tables) can be found in the file symlbin.c. The structures below are from the file symlbin.h.

```
extern struct symbolHashNode * HUGE_ADDR    *SymbolArray;
extern struct floatHashNode * HUGE_ADDR    *FloatArray;
extern struct integerHashNode * HUGE_ADDR    *IntegerArray;
extern struct bitMapHashNode * HUGE_ADDR    *BitMapArray;
```

```

#if FUZZY_DEFTEMPLATES
    extern struct fuzzyValueHashNode * HUGE_ADDR *FuzzyValueArray;
#endif

```

The structures below identify the additions/modifications necessary to save/restore information in **fuzzy deftemplates** into/from the binary file (see tmpltbin.h).

```

struct bsaveDeftemplate
{
    struct bsaveConstructHeader header;
    long slotList;
    unsigned int implied : 1;
    unsigned int numberOfSlots : 15;
    long patternNetwork;
#if FUZZY_DEFTEMPLATES
    unsigned int hasFuzzySlots : 1;
    long fuzzyTemplateList;
#endif
};

```

```

#if FUZZY_DEFTEMPLATES

```

```

struct bsaveLvPlusUniverse
{
    double from;
    double to;
    long unitsName;
    long ptPtr;
};

```

```

struct bsaveFuzzyPrimaryTerm
{
    long fuzzyValue;
    long next;
};

```

```

#endif

```

The code to perform this saving/loading of deftemplate information in a binary file is found in tmpltbin.c. The code to handle the fuzzy extensions is interspersed throughout the original code as required (enclosed in “if FUZZY_DEFTEMPLATES ... #endif”). Along with the added and modified structures above, 2 counters are defined:

```
static long NumberOfFuzzyTemplates;  
static long NumberOfFuzzyPrimaryTerms;
```

These counters are used to keep track of the number of FuzzyTemplates and the total number of Primary Term definitions (in all FuzzyTemplates).

Also there are 2 arrays that are used to store the restored structures when a BLOAD is done. These arrays are:

```
static struct fuzzyLv HUGE_ADDR *LvPlusUniverseArray;  
static struct primary_term HUGE_ADDR *PrimaryTermArray;
```

When a save is done all of the DefTemplate structures are written out by setting up the bsaveDefemplate structure for each deftemplate and then writing them to the binary file. If the template is a fuzzy one, the field fuzzyTemplateList is set to be an index into list of bsaveLvPlusUniverse structures that are written out after all deftemplates are written. If the deftemplate is not fuzzy then the field fuzzyTemplateList is set to -1. After the deftemplates are written out, the deftemplates are processed again to extract those that are fuzzy so that the bsaveLvPlusUniverse structures can be formed and written out consecutively. Then the deftemplates are processed one more time to get the bsaveFuzzyPrimaryTerm structures written out. After these are written then the standard (non-fuzzy) template slots for each template are written out to the binary file. The file will have a structure as shown below:

3 * Sizeof(long int)
NumberOfDeftemplates
NumberOfTemplateSlots
NumberOfTemplateModules
NumberOfFuzzyTemplates
NumberOfFuzzyPrimaryTerms

- not clear why this is included?

(the 3 is likely related to the number of "NumberOf" values that follow - before Fuzzy extensions this was 3; did not extend this to 5 because it doesn't appear to be used anywhere)

bytes needed for all structures
template module #1

template module #2

.
.

template module #NumberOfTemplateModules

deftemplate #1

deftemplate #2

.
.

deftemplate #NumberOfDeftemplates

LvPlusUniverse #1

LvPlusUniverse #2

.
.

LvPlusUniverse #NumberOfFuzzyTemplates

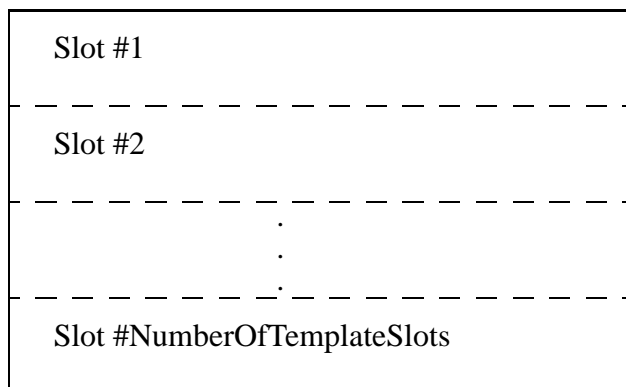
Primary Term #1

Primary Term #2

.
.

Primary Term #NumberOfPrimaryTerms

- includes deftemplates, template slots, fuzzy templates, primary terms, modifiers, and deftemplate modules.
- the fields fuzzyTemplateList in these structures are indices into the LvPlusUniverse structures stored in the next section
- the ptPtr (primary term ptr) entries in these structures are indices into the next set of structures; the modPtr (modifier ptr) entries are indices into the Modifier structures stored after the Primary Term Structures
- the **next** field of each structure is an index into this set of structures



There are quite a few details that need to be attended to in writing out these structures. For example when a symbol is referred to (as in the name of the UNITS in the universe specification), the bucket value is stored (this is an index into the SymbolArray).

A binary load is essentially a reverse of this writing operation where the arrays DeftemplateArray, LvPlusUniverseArray, PrimaryTermArray and SlotArray get constructed. The new routines UpdateLvPlusUniverse and UpdatePrimaryTerms handle this construction for the fuzzy template information. The code for these routines is not difficult to understand (the operation of the BloadandRefresh routine must also be understood).

10.0 Performing the Constructs-to-c Operation

The constructs-to-c routine is used to create a set of files that can be compiled with CLIPS (FuzzyCLIPS) to create a standalone runtime module (refer to the Section 5, pg 143 of the CLIPS Reference Manual, Vol. II, Advanced Programming Guide).

There are no new modules added to FuzzyCLIPS to support the constructs-to-c function. Instead existing code throughout CLIPS has been modified as necessary to create correct ascii files with the fuzzy extensions included. Files involved with changes are:

rulecmp.h	rulecmp.c
symlcmp.h	symlcmp.c
	tmpltcmp.c
	conscomp.c

The best way to understand the code is to see the output files for a simple example. Consider the following FuzzyCLIPS example (simplTst.clp):

```
; simple example with 2 rules that reason about how to change speed given
; an error rating of the speed
```

```
(deftemplate speed_error ; lv -- linguistic variable
  0 1 ; universe of discourse (range)
  ( ; linguistic term definitions
    (large_positive (0 0) (.1 .1) (.2 .2) (.3 .3) (.4 .4)
      (.5 .5) (.6 .6) (.7 .7) (.8 .8) (.9 .9) (1 1))
    (zero (0 1) (.11 0))
    (small_positive (0 1) (1 0))
  )
)
```

```
(deftemplate speed_change ; lv -- linguistic variable
  0 1 ; universe of discourse (range)
  ( ; linguistic term definitions
    (large_negative (.1 0) (.2 .2) (.3 .4) (.4 .6) (.5 .8) (.6 1))
    (none (0 1) (.1 .1) (.2 0))
    (small_negative (.4 1) (.5 .8) (.6 .6) (.7 .4) (.8 .2) (.9 0))
  )
)
```

```
(deffacts my_facts
  (speed_error zero) CF .9
)
```

```
; NOTE: both of these next 2 rules have a contribution to make to the solution
; resulting in a fuzzy set for speed-change that reflects this
```

```
;      combination after both rules have fired

(defrule speed-too-fast ; type fuzzy-fuzzy
  (declare (CF .7))

  (speed_error large_positive)

=>
  (assert (speed_change large_negative))
)

(defrule speed-ok ; type fuzzy-fuzzy
  (declare (CF .7))

  (speed_error zero)

=>
  (assert (speed_change none))
)

; the next rule takes the result of the previous 2 rules and produces a non-fuzzy
; result to identify the amount (between 0 and 1) to change the speed
; and finally print out the de-fuzzified result

(defrule get-crisp-value-and-print-result
  (declare (salience -1))
  ?sc <- (speed_change ?)

=>
  (bind ?f (moment-defuzzify ?sc))
  (printout t "Change speed by a factor of: " ?f crlf)
)
```

The example is loaded and then the constructs-to-c function is called as

```
(constructs-to-c smp 1)
```

A large number of files are created to reflect the 'C' code needed to define the has tables, constructs, etc. that define this program and allow a runtime version to be created. The main file smp.c is generated as follows:

```
#include "smp.h"

#include "utility.h"
#include "generate.h"
#include "expressn.h"
#include "extnfunc.h"
```

```

#include "objrtmch.h"
#include "rulebld.h"

/*****
/* CONSTRUCT IMAGE INITIALIZATION FUNCTION */
*****/

VOID InitCImage_1()
{
    Clear();
    PeriodicCleanup(CLIPS_TRUE,CLIPS_FALSE);
    SetSymbolTable(sht1);
    SetFloatTable(fht1);
    SetIntegerTable(iht1);
    SetBitMapTable(bmht1);
    SetFuzzyValueTable(fvht1);
    RefreshBooleanSymbols();
    InstallFunctionList(P1_1);

    InitExpressionPointers();

    SetListOfDefmodules((VOID *) Y_1_1);
    SetCurrentModule((VOID *) GetNextDefmodule(NULL));

    SetObjectNetworkPointer(NULL);
    SetObjectNetworkTerminalPointer(NULL);

    ObjectsRunTimeInitialize(S_1_1,R_1_1,T_1_1,U_1);

    ResetDefglobals();
}

```

The name `InitCImage_1()` is determined by the integer 1 provided to the constructs-to-c function when the code was generated. Note that all of the names of the hash table have the value 1 appended to them as well. For example, the symbol hash table is referenced as 'sht1' and the fuzzy value hash table as 'fvht1'.

The file `conscomp.c` is where the constructs-to-c function is defined. It checks arguments and calls the `GenerateCode` function with the filename arg (in our example 'smp') and the `ImagID` (in our example 1). This function in turn writes out the header file (`cmp.h`), and write out other files that define the constructs and expressions used in this program. The last thing `GenerateCode` creates is the `smp.c` file shown above (via the call to `SetUpInit-File(fileName)`). We will not explain all of the files created by this process but will describe the process as it relates to the Fuzzy extensions.

The call to AtomicValuesToCode in GenerateCode causes files to be generated for all atomic items (symbols, floats, integers, FuzzyValues, etc.). The code for this is found in symblcmp.c. Before looking at the tables created for FuzzyValues let's look at the code for the symbol table. In the HeaderFP (smp.h) by calling the routine HashTablesToCode it adds the lines:

```
/******  
/* EXTERN ARRAY DEFINITIONS */  
/******  
  
extern struct symbolHashNode *sht1[];  
extern struct floatHashNode *fht1[];  
extern struct integerHashNode *iht1[];  
extern struct bitMapHashNode *bmht1[];  
extern struct fuzzyValueHashNode *fvht1[];
```

During the call to HashTablesToCode it also creates a file smp1_1.c with the following contents (this is the symbol hash table):

```
#include "smp.h"  
  
struct symbolHashNode *sht1[1013] = {  
    NULL,  
    NULL,  
    &S1_1[0],  
    NULL,  
    &S1_1[2],  
    &S1_1[4],  
    NULL,  
    &S1_1[5],  
    &S1_1[7],  
    NULL,  
    NULL,  
    &S1_1[8],  
    &S1_1[10],  
    NULL,  
    &S1_1[11],  
    &S1_1[12],  
    NULL,  
    &S1_1[13],  
    NULL,  
    NULL,  
    NULL,  
    &S1_1[15],
```

```

NULL,
NULL,
NULL,
.
.
.
NULL,
NULL,
&S1_1[438],
NULL,
NULL,
NULL,
&S1_1[441],
&S1_1[442],
NULL,
&S1_1[443],
NULL,
&S1_1[444],
&S1_1[446]};

```

Note that this references an array `&S1_1` which actually holds the hash nodes for the symbol table. The indices are obtained from the bucket field of the hash nodes. These values were set by a previous call to `SetAtomicValueIndices` (see `symbol.c`) just before `HashTablesToCode` was called. Essentially what `SetAtomicValueIndices` does is walk through each hash table and set the bucket slot to be a consecutive integer (from 0 to the number of atomic items in the hash table - 1) for each item in the hash table. After `HashTablesToCode` is done then the individual atoms (hash nodes) are written to their own files in an array. For example the file `smp1_6.c` defines the hash nodes for the symbol hash table `S1_1` as follows:

```

#include "smp.h"

struct symbolHashNode S1_1[] = {
  {&S1_1[1],2,0,0,0,2,"-"},
  {NULL,2,0,0,0,2,"message-duplicate-instance"},
  {&S1_1[3],2,0,0,0,4,"slot-types"},
  {NULL,2,0,0,0,4,"put-name"},
  {NULL,2,0,0,0,5,"get-u"},
  {&S1_1[6],2,0,0,0,7,"tanh"},
  {NULL,2,0,0,0,7,"defclass-module"},
  {NULL,2,0,0,0,8,"instance-address"},
  {&S1_1[9],2,0,0,0,11,"toss"},
  .
  .
  .

```

```
{NULL,2,0,0,0,987,"list-defglobals"},
{NULL,2,0,0,0,988,"neq"},
{NULL,2,0,0,0,989,"list-deftemplates"},
{NULL,2,0,0,0,990,"INITIAL-OBJECT"},
{NULL,2,0,0,0,992,"(query-instance)"},
{NULL,2,0,0,0,993,"(direct-duplicate)"},
{NULL,2,0,0,0,994,"run"},
{NULL,2,0,0,0,996,"(message-modify)"},
{&S1_1[439],2,0,0,0,1002,"str-implode"},
{&S1_1[440],2,0,0,0,1002,"str-explode"},
{NULL,2,0,0,0,1002,"retract"},
{NULL,2,0,0,0,1006,"sec"},
{NULL,2,0,0,0,1007,"format"},
{NULL,2,0,0,0,1009,"focus"},
{&S1_1[445],2,0,0,0,1011,"unwatch"},
{NULL,2,0,0,0,1011,"get-fs-lv"},
{NULL,2,0,0,0,1012,"undefclass"}};
```

The table S1_1 for symbol hash nodes has references (pointers) to itself in the ‘next’ field of the nodes. This represents a linked list of symbols that hash to the same symbol table entry.

For the FuzzyValue has table a similar set of files are created. The file `smp1_5.c` defines the FuzzyValue has table:

```
#include "smp.h"
```

[illegible]

[illegible]

The file `smp1_11.c` holds the fuzzyValue hash nodes(table V1_1):

```
#include "smp.h"
```

```
struct fuzzyValueHashNode V1_1[] = {
{NULL,2,0,0,0,4,(struct fuzzy_value *) &W1_1[0]},      ; “small_positive”
{NULL,4,0,0,0,20,(struct fuzzy_value *) &W1_1[1]},      ; “zero”
{NULL,3,0,0,0,65,(struct fuzzy_value *) &W1_1[2]},      ; “none”
{&V1_1[4],3,0,0,0,93,(struct fuzzy_value *) &W1_1[3]},  ; “large_positive”
{NULL,3,0,0,0,93,(struct fuzzy_value *) &W1_1[4]},      ; “large_negative”
{NULL,2,0,0,0,150,(struct fuzzy_value *) &W1_1[5]}};    ; “small_negative”
```

Because the FuzzyValues are complex structures themselves they are not stored directly in the V1_1 table. Two other files define 2 other tables (W1_1 and X1_1) that complete the definition of the fuzzyValues. The file `smp1_12.c` defines the W1_1 table:

```
#include "smp.h"
```

```
struct fuzzy_value W1_1[] = {
  {&C_1_1[1],&S1_1[260],2,2,(double *)&X1_1[0], (double *)&X1_1[2]},
  {&C_1_1[1],&S1_1[123],2,2,(double *)&X1_1[4], (double *)&X1_1[6]},
  {&C_1_1[2],&S1_1[121],3,3,(double *)&X1_1[8], (double *)&X1_1[11]},
  {&C_1_1[1],&S1_1[358],11,11,(double *)&X1_1[14], (double *)&X1_1[25]},
  {&C_1_1[2],&S1_1[354],6,6,(double *)&X1_1[36], (double *)&X1_1[42]},
  {&C_1_1[2],&S1_1[253],6,6,(double *)&X1_1[48], (double *)&X1_1[54]};
```

where each entry describes a fuzzy_value structure. Take the entry

```
{&C_1_1[1],&S1_1[260],2,2,(double *)&X1_1[0], (double *)&X1_1[2]},
```

The first field, &C_1_1[1], is the reference to the fuzzy deftemplate associated with this fuzzy value. The second field, &S1_1[260], is a pointer to a symbol hash node that holds the FuzzyValue's name. The third field, 2, is the maxn value and the fourth field, 2, is the n value. The fifth field points to the array of x values for the fuzzy set (in table X1_1) and the sixth value points to the array of y values for the fuzzy set (in table X1_1). The X1_1 table is found in smp1_13.c as follows:

```
#include "smp.h"
```

```
double X1_1[] = {
0.0,1.0,
1.0,0.0,
0.0,0.11,
1.0,0.0,
0.0,0.1,0.2,
1.0,0.1,0.0,
0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0,
0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0,
0.1,0.2,0.3,0.4,0.5,0.6,
0.0,0.2,0.4,0.6,0.8,1.0,
0.4,0.5,0.6,0.7,0.8,0.9,
1.0,0.8,0.6,0.4,0.2,0.0};
```

Therefore X1_1[0] = 0.0 and X1_1[1] = 1.0 define the x array and X1_1[2] = 1.0 and X1_1[3] = 0.0 define the y array.

Note that the beginning of the file conscomp.c defines the list of prefix characters used to define these tables; S for the Symbol Hash node, V for the fuzzy value hash nodes, etc. The routines FuzzyValueHashNodesToCode, FuzzyValuesToCode, and FuzzyValueArraysToCode (in symblcmp.c) are called by the routine AtomicValuesToCode (in symblcmp.c) to create these tables. The code is straight forward once the structure of the hash tables is understood and the structure of fuzzy_values is understood.

After AtomicValuesToCode is completed we are back in routine GenerateCode (file conscomp.c). Routines FunctionsToCode, HashedExpressionsToCode and ConstraintsToCode are called. These have little to do with the fuzzy extensions except for the routine HashedExpressionsToCode, which can while writing out hashed expression encounter an expr node of type FUZZY_VALUE. In this case a FuzzyValue has node reference is placed in the file (reference to V1_1 table -- see DumpExpression in file conscomp.c and PrintFuzzyValueReference in file symbicmp.c).

The next piece of code in GenerateCode deals with calling functions to generate code for the constructs such as deffacts, deftemplates and defrules. Of interest to the fuzzy extensions are the generation of code for rules and templates.

The routine ConstructToCode in rulecmp.c takes care of generating the code for defrule constructs. The code generation for a defrule is similar to the original code provided by CLIPS except that the extra fields are also generated and the field pattern_fv_arrayPtr which points to an array of fuzzy slot locator structures is handled by creating a special file with a table of all of these pointers stored in them. The file smp15_2.c has the defrule structures for the 3 rules of the example:

```
#include "smp.h"
```

```
struct defrule D1_1[] = {
  {{&S1_1[202],NULL,MIHS &A1_1[0],0,CHS
  &D1_1[1]},0,0,2,0,0,0,0,0,NULL,&E1_1[25],NULL,&G1_1[0],NULL,0.7,&E1_1[29],-
  1.0,220,1,&H1_1[0]},
  {{&S1_1[344],NULL,MIHS &A1_1[0],0,CHS
  &D1_1[2]},0,0,2,0,0,0,0,0,NULL,&E1_1[30],NULL,&G1_1[1],NULL,0.7,&E1_1[34],-
  1.0,220,1,&H1_1[1]},
  {{&S1_1[292],NULL,MIHS &A1_1[0],0,NULL},-
  1,1,1,0,0,0,0,0,NULL,&E1_1[35],NULL,&G1_1[2],NULL,1.0,NULL,-
  1.0,219,0,NULL}};
```

Taking the 1st rule definition we see that the fields are:

```
name          = &S1_1[202]          ; construct header
ppForm        = NULL
whichModule   = &A1_1[0]
bsaveID       = 0
next          = &D1_1[1]

salience     = 0
localVarCnt   = 0
complexity    = 2
afterBreakpoint = 0
watchActivation = 0
```

```
watchFiring    = 0
autoFocus      = 0
executing      = 0
dynamicSalience= NULL
actions        = &E1_1[25]
logicalJoin    = NULL
lastJoin       = &G1_1[0]
disjunct       = NULL
CF             = 0.7
dynamicCF      = &E1_1[29]
min_of_maxmins= -1.0
lhsRuleType    = 220 (FUZZY_LHS)
numberOfPatterns= 1
pattern_fv_arrayPtr= &H1_1[0]    ; see H1_1 defined below
```

The pattern_fv_arrayPtr pointers are define in file smp15_3.c as follows:

```
#include "smp.h"

struct fzSlotLocator H1_1[] = {
{0, 0, &V1_1[3]},      ; fuzzy value "large_positive"
{0, 0, &V1_1[1]}};    ; fuzzy value "zero"
```

Again the code is quite straight forward once the output requirements are understood.

The routine ConstructToCode in tmpltcmp.c takes care of generating the code for deftemplate constructs. The code generation for a deftemplate is similar to the original code provided by CLIPS except for the addition of hasFuzzySlots field and the fuzzyTemplate field that must be written out for fuzzy deftemplates and the associated structures pointed to by the latter field. The extra data to be written due to the fuzzyLv struct pointed to by fuzzyTemplate is handled by creating 3 new files. First let's look at the basic deftemplate structures written out in the file smp9_2.c:

```
#include "smp.h"

struct deftemplate C_1_1[] = {
{{&S1_1[176], NULL,MIHS &B_1_1[0],0, CHS &C_1_1[1]},
 NULL,0,0,0,0,1,NULL,0,NULL},
{{&S1_1[244], NULL,MIHS &B_1_1[0],0, CHS &C_1_1[2]},
 &D_1_1[0],0,0,0,1,4,&A_1_1[0],1,(struct fuzzyLv *)&E_1_1[0]},
{{&S1_1[101], NULL,MIHS &B_1_1[0],0, NULL},
 &D_1_1[1],0,0,0,1,4,&A_1_1[2],1,(struct fuzzyLv *)&E_1_1[1]}};
```

The first entry is the “initial-fact” deftemplate. The next 2 are for the “speed_error” and “speed_change” deftemplates (both fuzzy). The last field of each of these is of interest for fuzzy extensions since it is a pointer to a list of pointers to fuzzyLv structures. The E_1_1 array is defined in smp9_3.c:

```
#include "smp.h"

struct fuzzyLv E_1_1[] = {
  {0.0, 1.0, NULL, &F_1_1[0] },
  {0.0, 1.0, NULL, &F_1_1[3] } };
```

The entry &F_1_1[0] refers to a list of terms for the fuzzy value. File smp9_4.c defines this list of fuzzy terms in F_1_1. Each entry in F_1_1 is a node in the list of terms that includes a pointer to a fuzzy_value hash node and the next ptr to next term in the list (i.e. a reference into the F_1_1 table):

```
#include "smp.h"

struct primary_term F_1_1[] = {
  {&V1_1[3],&F_1_1[1]},{&V1_1[1],&F_1_1[2]},{&V1_1[0],NULL},
  {&V1_1[4],&F_1_1[4]},{&V1_1[2],&F_1_1[5]},{&V1_1[5],NULL}};
```

The 1st entries in this F_1_1 table constitute one linked list terminated by a NULL pointer and the next 2 are a second linked list of 2 nodes terminated by a NULL pointer.

The structures for this are somewhat more complicated than others used in the code created by constructs-to-c and it will be necessary to understand the example in detail before the code will be easily read or modified.

*** NOTE WELL ***

Not described at all well are the ‘constraints’ structures as modified for Fuzzy Value type checking. This is partly because it is likely not very well implemented/integrated with standard CLIPS -- i.e. some work may be needed in this area.

Also note that there is no substitute for reading the code to see what is really done!!!!