Appendix A

Brief Outline of CLIPS

A.1 Data Types

CLIPS provides primitive data types for representing information including *float*, *integer*, *symbol*;*string*,*fact–address*. Numeric information can be represented using floats and integers. Symbolic information can be represented using symbols and strings.

A.2 Functions

A *function* in CLIPS is a piece of executable code identified by a specific name which returns a useful value or performs a useful side effect (such as displaying information).

The *deffunction* construct allows users to define new functions directly in the CLIPS environment using CLIPS syntax. They are interpreted by the CLIPS environment.

Function calls in CLIPS use prefix notation — the arguments to a function always appear after the function name. Function calls begin with a left parenthesis, followed by the name of the function, then the arguments to the function follow (each argument separated by one or more spaces). Examples of function calls using the addition (+) and multiplication (*) functions are shown following.

```
(+ 3 4 5)
(* 5 6.0 2)
(+ 3 (* 8 9) 4)
(* 8 (+ 3 (* 2 3 4) 9) (* 3 4))
```

A.3 Constructs

Several defining *constructs* appear in CLIPS including: *defrule*, *deffacts*, *deftemplate* and *deffunction*. The construct opens with a left parenthesis and closes with a right parenthesis. Defining a construct differs from calling a function primarily in effect.

Constructs allow a comment directly following the construct name. Comments also can be placed within CLIPS code by using a semicolon. Everything from the ; until the next return character will be ignored by CLIPS.

A.4 Deffacts Construct

A list of facts can be defined which are automatically asserted whenever the *reset* command is performed. Facts asserted through deffacts may be retracted or pattern matched like any other fact.t; command.
Syntax

```
(deffacts <deffacts-name>
   [<comment>]       <RHS-pattern>*)
```

Example

```
(deffacts startup "Refrigerator Status"
     (refrigerator light on)
     (refrigerator door open)
     (refrigerator temp (get-temp)))
```

A.5 Defrule Construct

A *rule* is a collection of conditions and the actions to be taken if the conditions are met. Rules execute (or *fire*) based on the existence or non–existence of facts. CLIPS provides the mechanism (the *inference engine*) which attempts to match the rules to the current state of the system (as represented by the *fact–list* or *working memory*) and applies the actions.

The *defrule* construct:

```
(defrule <rule-name> [<comment>]
   [<declaration>]               ; Rule Properties
   <conditional-element>*        ; Left-Hand Side (LHS)
   =>
   <action>*)                    ; Right-Hand Side (RHS)
```

A.6 BASIC CYCLE OF RULE EXECUTION

Once a knowledge base (in the form of rules) is built and the fact-list and instance-list is prepared, CLIPS is ready to execute rules. In a conventional language, the starting point, the stopping point, and the sequence of operations are defined explicitly by the programmer. With CLIPS, the program flow does not need to be defined quite so explicitly.

The knowledge (rules) and the data (facts and instances) are separated, and the inference engine provided by CLIPS is used to apply the knowledge to the data. The basic execution cycle is as follows:

a) If the rule firing limit has been reached or there is no current focus, then execution is halted. Otherwise, the top rule on the agenda of the module which is the current focus is selected for execution. If there are no rules on that agenda, then the current focus is removed from the focus stack and the current focus becomes the next module on the focus stack. If the focus stack is empty, then execution is halted, otherwise step a is executed again.

b) The right-hand side (RHS) actions of the selected rule are executed. The use of the return function on the RHS of a rule may remove the current focus from the focus stack. The number of rules fired is incremented for use with the rule firing limit.

c) As a result of step b, rules may be *activated* or *deactivated*. Activated rules (those rules whose conditions are currently satisfied) are placed on the agenda. The placement on the agenda is determined by the salience of the rule and the current conflict resolution strategy. Deactivated rules are removed from the agenda. If the activations item is being watched, then an informational message will be displayed each time a rule is activated or deactivated.

d) If dynamic salience is being used, the salience values for all rules on the agenda are re–evaluated. Repeat the cycle beginning with step a.

A.7 CONFLICT RESOLUTION STRATEGIES

The agenda is the list of all rules which have their conditions satisfied (and have not yet been executed). The agenda acts similar to a stack (the top rule on the agenda is the first one to be executed). When a rule is newly activated, its placement on the agenda is based (in order) on the following factors:

a) Newly activated rules are placed above all rules of lower salience and below all rules of higher salience.

b) Among rules of equal salience, the current conflict resolution strategy is used to determine the placement among the other rules of equal salience.

c) If a rule is activated (along with several other rules) by the same assertion or retraction of a fact, and steps a and b are unable to specify an ordering, then the rule is arbitrarily (not randomly) ordered in relation to the other rules with which it was activated. Note, in this respect, the order in which rules are defined has an arbitrary effect on conflict resolution (which is highly dependent upon the current underlying implementation of rules). Do not depend upon this arbitrary ordering for the proper execution of your rules.

CLIPS provides seven conflict resolution strategies: *depth*, *breadth*, *simplicity*, *complexity*, *lex*, *mea*, and *random*. The default strategy is depth. The current strategy can be set by using the set-strategy command (which will reorder the agenda based upon the new strategy).

A.7.1 Depth Strategy

Newly activated rules are placed above all rules of the same salience. For example, given that fact-a activates rule-1 and rule-2 and fact-b activates rule-3 and rule-4, then if fact-a is asserted before fact-b, rule-3 and rule-4 will be above rule-1 and rule-2 on the agenda. However, the position of rule-1 relative to rule-2 and rule-3 relative to rule-4 will be arbitrary.

A.7.2 Breadth Strategy

Newly activated rules are placed below all rules of the same salience. For example, given that fact-a activates rule-1 and rule-2 and fact-b activates rule-3 and rule-4, then if fact-a is asserted before fact-b, rule-1 and rule-2 will be above rule-3 and rule-4 on the agenda. However, the position of rule-1 relative to rule-2 and rule-3 relative to rule-4 will be arbitrary.

A.7.3 Simplicity Strategy

Among rules of the same salience, newly activated rules are placed above all activations of rules with equal or higher *specificity*. The specificity of a rule is determined by the number of comparisons that must be performed on the LHS of the rule. Each comparison to a constant or previously bound variable adds one to the specificity. Each function call made on the LHS of a rule as part of the :, =, or test conditional element adds one to the specificity. The boolean functions **and**, **or**, and **not** do not add to the specificity of a rule, but their arguments do. Function calls made within a function call do not add to the specificity of a rule. For example, the following rule

```
(defrule example
   (item ?x ?y ?x)
   (test (and (numberp ?x) (> ?x (+ 10 ?y)) (< ?x 100)))
   =>)
```

has a specificity of 5. The comparison to the constant item, the comparison of ?x to its previous binding, and the calls to the numberp, <, and > functions each add one to the specificity for a total of 5. The calls to the and and + functions do not add to the specificity of the rule.

A.7.4 Complexity Strategy

Among rules of the same salience, newly activated rules are placed above all activations of rules with equal or lower specificity.

A.7.5 LEX Strategy

This is a technical strategy, for information see the reference manual.

A.7.6 MEA Strategy

This is a technical strategy, for information see the reference manual.

A.7.7 Random Strategy

Each activation is assigned a random number which is used to determine its placement among activations of equal salience. This random number is preserved when the strategy is changed so that the same ordering is reproduced when the random strategy is selected again (among activations that were on the agenda when the strategy was originally changed).

A.8 Constraints

A.8.1 Connective Constraints

Three *connective constraints* are &, | and ∼. The & constraint is satisfied if the two adjoining constraints are satisfied. The | constraint is satisfied if either of the two adjoining constraints is satisfied and ∼ constraint is satisfied if the following constraint is not satisfied. The connective constraints can be combined.

```
; data-B can have a slot called value
(deftemplate data-B
(slot value))

(deffacts AB
  (data-A green)
  (data-A blue)
  (data-B (value red))
  (data-B (value blue)))

(defrule example1-1
  (data-A ~blue)
  =>)
```

```
(defrule example1-2
  (data-B (value ~red&~green))
  =>)

(defrule example1-3
  (data-B (value green|red))
  =>)
CLIPS> (reset)
facts are
f-0     (initial-fact)
f-1     (data-A green)
f-2     (data-A blue)
f-3     (data-B (value red))
f-4     (data-B (value blue))

rule agenda
0      example1-2: f-4
0      example1-3: f-3
0      example1-1: f-1
```

A.8.2 Predicate Constraints

To constrain a field based upon the truth of a given boolean expression. The predicate constraint allows a predicate function to be called during the pattern matching process.

```
(defrule example-1
  (data ?x&:(numberp ?x))
  =>)
; constrains ?x to be number - read 'such that'

(defrule example-2
  (data ?x&~:(symbolp ?x))
  =>)
; ?x must not be a symbol

(defrule example-3
  (data ?x&:(numberp ?x)&:(oddp ?x))
  =>)
; ?x must be a number and be odd

(defrule example-4
  (data ?y)
  (data ?x&:(> ?x ?y))
```

```
  =>)
; ?x must be > ?y
```

A.8.3 Pattern Addresses

A variable can be bound to a *fact-address* using $< -$. This can then be used for retracting a fact.

```
(defrule dummy
   (data 1)
   ?fact <- (dummy pattern)
   =>
   (retract ?fact))
; pick up fact then retract
```

A.8.4 Test

General constraints as premises of rules have to be evaluated using *test*.

```
(defrule example-1
  (data ?x)
  (value ?y)
  (test (>= (abs (- ?y ?x)) 3))
  =>)
; make sure |?x -?y| >= 3
```

A.8.5 Boolean connectives

Premises can be combines using the connectives *and*, *or* and *not*.

```
(defrule system-flow
    (error-status confirmed)
    (or (and (temp high)
             (valve closed))
        (and (temp low)
             (valve open)))
    =>
   (printout t "The system is having a flow problem." crlf))
```

A.9 Salience Declaration

Salience declaration allows the user to assign a priority to a rule. When multiple rules are in agenda, the rule with the highest priority will fire first. Salience is an integer in the range -10000 to +10000 which defaults to zero.

```
(defrule test-1
   (declare (salience 99))
   (fire test-1)
   =>
   (printout t "Rule test-1 firing." crlf))

(defrule test-2
   (declare (salience (+ ?*constraint-salience* 10)))
; can be evaluated
   (fire test-2)
   =>
   (printout t "Rule test-2 firing." crlf))
```

By default, salience values are only evaluated when a rule is defined but this can be changed.

A.10 Defglobal Construct

Globals, such as ?*x*, can be defined using *defglobal*

```
(defglobal
   ?*x* = 3
   ?*y* = ?*x*
   ?*z* = (+ ?*x* ?*y*)
   ?*q* = (create$ a b c))
```

A.11 Deffunction Construct

Syntax

```
(deffunction <name> [<comment>]
(<regular-parameter>* [<wildcard-parameter>])
<action>*)

<regular-parameter>::= <single-field-variable>
<wildcard-parameter>::= <multifield-variable>

(deffunction factorial (?a)
(if (or (not (integerp ?a)) (< ?a 0)) then
(printout t "Factorial Error!" crlf)
 else
```

```
(if (= ?a 0) then
1
 else
(* ?a (factorial (- ?a 1))))))
```

A.12 Actions and Functions

A.12.1 Predicate Functions

Testing For Numbers

`(numberp <expression>)`

Syntax

Testing for Floats

`(floatp <expression>)`

Testing for Integers

`(integerp <expression>)`

Testing for Strings or Symbols

TRUE if its argument is a string or symbol.

`(lexemep <expression>)`

Testing for Strings

`(stringp <expression>)`

Testing for Symbols

`(symbolp <expression>)`

Testing for Even/Odd Numbers

`(evenp <expression>)` `(oddp <expression>)`

Comparing for Equality/Inequality

`(eq <expression> <expression>+)` `(neq <expression> <expression>+)`

Comparing Numbers for Equality/Inequality

```
(= <numeric-expression> <numeric-expression>+)
(<> <numeric-expression> <numeric-expression>+)
```

Greater Than/Greater Than or Equal

```
(> <numeric-expression> <numeric-expression>+)
(>= <numeric-expression> <numeric-expression>+)
```

Less Than/Less Than or Equal

```
(< <numeric-expression> <numeric-expression>+)
(<= <numeric-expression> <numeric-expression>+)
```

Boolean Functions

```
(and <expression>+)
(or <expression>+)
(not <expression>)
```

A.12.2 String Functions

String Concatenation

```
(str-cat <expression>*)
```

Symbol Concatenation

Concatenate its arguments into a single symbol.

```
(sym-cat <expression>*)
```

Taking a String Apart

```
(sub-string <integer-expression> <integer-expression>
         <string-expression>)

(sub-string 3 8 "abcdefghijkl")
"cdefgh"
```

Searching a String

Return the position of a string inside another string.

```
(str-index  <lexeme-expression> <lexeme-expression>)

(str-index "def" "abcdefghi")
answer 4
```

Converting a String to Uppercase/Lowercase

```
(upcase <string-or-symbol-expression>)

(upcase "This is a test of upcase")
answer "THIS IS A TEST OF UPCASE"

(lowcase <string-or-symbol-expression>)
```

Comparing Two Strings

str-compare compares two strings. Comparison is performed character by character until the strings are exhausted (implying equal strings) or un-equal characters are found. The positions of the unequal characters within the ASCII character set are used to determine the logical relationship of unequal strings.

```
(str-compare <string-or-symbol-expression>
             <string-or-symbol-expression>)
```

This function returns an integer representing the result of the comparison (0 if the strings are equal, < 0 if the first argument $<$ the second argument, and > 0 if the first argument $>$ the second argument).

```
(< (str-compare "string1" "string2") 0) TRUE
 ; since "1" < "2" in ASCII character set
    (str-compare "abcd" "abcd")  0
```

Length of a String

```
(str-length <string-or-symbol-expression>)
(str-length "abcd")    4
```

A.12.3 CLIPS I/O

Printout

Output to a device attached to a logical name. The logical name must be specified and the device must have been prepared previously for output (e.g., a file must be opened first). To send output to stdout use t for the logical name.

```
(printout <logical-name> <expression>*)

(printout t "Hello there!" crlf)
Hello There!

(open "data.txt" mydata "w")
(printout mydata "red green")
(close)
```

Read

Allows a user to input information for a single field.

```
(read [<logical-name>])
```

where <logical-name> is an optional parameter. If specified, read tries to read from whatever is attached to the logical file name.

A.12.4 Mathematical Functions

Addition/Subtraction/Multiplication/Division

```
(+ <numeric-expression> <numeric-expression>+) etc
(+ 2 3 4)
9

(- 12 3 4)
5

(* 2 3 4)
24
(/ 4 2)
2.0
(/ 24 3 4)
2.0
```

Integer Division

```
(div <numeric-expression> <numeric-expression>+)
(div 33 2 3 5)
1
```

Maximum/Minimum

```
(max <numeric-expression>+)
(max 3.0 4 2.0)
 4
(min 4 0.1 -2.3)
 -2.3
```

Absolute Value

```
(abs <numeric-expression>)
(abs 4.0)
 4.0
```

Convert to Float/Integer

```
(float <numeric-expression>)
(float -2)
 -2.0

(integer <numeric-expression>)
(integer 4.0)
 4
```

A.12.5 Extended Maths Functions

There are many more functions such as trigonometric, exponential, hyperbolic etc. They usually have the expected names.

A.12.6 Procedural Functions

Procedural programming capabilities given by:

Binding Variables

```
     (bind <variable> <expression>*)
(bind ?x 2)
```

If...then...else Function

```
(if <expression>
then <action>
[else <action>*])
```

```
(defrule closed-valves
(temp high)
(valve ?v closed)
 =>
(if (= ?v 6)
then
  (printout t "The special valve " ?v " is closed!" crlf)
  (assert (perform special operation))
else
  (printout t "Valve " ?v " is normally closed"crlf)))
```

While

```
(while <expression> [do]
  <action>*)
```

```
(defrule open-valves
(valves-open-through ?v)
=>
(while (> ?v 0)
       (printout t "Valve " ?v " is open" crlf)
(bind ?v (- ?v 1))))
```

Loop-for-count

```
(loop-for-count <range-spec> [do] <action>*)
```

```
(loop-for-count 2 (printout t "Hello world" crlf))
```

```
Hello world
Hello world
```

```
(loop-for-count (?cnt1 2 4) do
 (printout t ?cnt1 crlf)))
2
3
4
```

Return

```
(return [<expression>])
(deffunction sign (?num)
  (if (> ?num 0) then
    (return 1))
 (if (< ?num 0) then
    (return -1))
 0)

(sign -10)
-1
```

Switch

```
(switch <test-expression>
  <case-statement>
  <case-statement>+
  [<default-statement>])

(deffunction foo (?val)
   (switch ?val
   (case 1 then mon)
       (case 2 then tue)
       (default none)))
```

A.12.7 Fact Functions

Creating New Facts

```
(assert <RHS-pattern>+)

(assert (color red))
(assert (color blue) (value (+ 3 4)))

(facts)
f-0     (color red)
f-1     (color blue)
f-2     (value 7)
```

Removing Facts from the Fact-list

```
(retract <retract-specifier>+ | *)
```

```
(defrule change-valve-status
    ?f1 <- (valve ?v open)
    ?f2 <- (set ?v close)
 =>
    (retract ?f1 ?f2)
    (assert (valve ?v close)))
```

A.13 CLIPS Software

You can obtain software and documentation from CMU AI repository. The WWW URL is

`http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/expert/systems/clips/0.html`