Lecture 3 and lab 3

3.1 How to redirect the output to a file

Instead of the output being displayed in the output window, we can direct the output stream to a file with the option "**Turn Dribble On…"** from the File menu. In this case we will be asked about the name of the output file, and the location in a dialog window.
The second variant to do the same thing is with the command (dribble-on <file-name>).

3.2 How to define type restrictions and default values to fields from a template

We can restrict the type of the fields of a template construct to specific data types, also we can define default values for the fields as in the following example:

```
(deftemplate prospect ;name of deftemplate relation
"vital information" ;optional comment in quotes
  (slot name        ;name of field
     (type STRING)       ;type of field
     (default ?DERIVE))  ;default value of field name
  (slot assets           ;name of field
     (type SYMBOL)       ;type of field
     (default rich))     ;default value of field assets
  (slot age              ;name of field
     (type NUMBER)       ;type. NUMBER can be INTEGER or FLOAT
     (default 80))       ;default value of field age
)
```

For the previous example, if no value is supplied for a certain field, then the default value is used. This happened when we issue the (reset) command.

It is also possible to say, for example for slot age the type to be INTEGER or FLOAT, in the manner presented below, thus we can see that we can enumerate more than one type for a field. Theres another thing, also to be specified, NUMBER is not a primitive data type, it is composed of the subtypes INTEGER and FLOAT.

```
(slot age
(type INTEGER FLOAT)
(default 80)))
```

```
The allowed values for the primitive types can be seen from the
table below, also you can see another restriction that we can
impose on the values of the fields:,
```

```
Deftemplate Enumerated Values Example
allowed-symbols rich filthy-rich loaded
allowed-strings "Dopey" "Dorky" "Dicky"
allowed-numbers 1 2 3 4.5 -2.001 1.3e-4
allowed-integers -100 53
allowed-floats -2.3 1.0 300.00056
allowed-values "Dopey" rich 99 1.e9
```

Example:

```
(deftemplate prospect
     (multislot name
          (type SYMBOL)
          (default ?DERIVE))
     (slot assets
          (type SYMBOL)
          (allowed-symbols poor rich wealthy loaded)
          (default rich))
     (slot age
          (type INTEGER)
          (range 80 ?VARIABLE) ; The older the better!!!
          (default 80))
)
```

Refference ug.pdf pages 53-61

3.3 Working with multifield slots.

We can create a multifield slot with the use of the create$ function with the syntax

(create$ <arg1> <arg2>...<argN>)

An example is presented below:

```
CLIPS> (defrule bind-values-demo
     (initial-fact)
     =>
     (bind ?duck-bachelors (create$ Dopey Dorky Dinky))
     (bind ?happy-bachelor-mv (create$ Dopey))
     (bind ?none (create$))
     (printout t
     "duck-bachelors " ?duck-bachelors crlf
     "duck-bachelors-no-() " (implode$ ?duck-bachelors) crlf
     "happy-bachelor-mv " ?happy-bachelor-mv crlf
     "none " ?none crlf))
CLIPS> (reset)
CLIPS> (run)
duck-bachelors (Dopey Dorky Dinky)
duck-bachelors-no-() Dopey Dorky Dinky
happy-bachelor-mv (Dopey)
none ()
```

Some other useful functions that can be used are:

round Round toward closest integer. If exactly between two integers, rounds toward negative infinity.
integer Truncates the decimal part of a number.
format Formats output
list-deffunctions List all deffunctions
ppdeffunction Pretty print
undeffunction Deletes a deffunction if it is not currently executing and not referred to elsewhere. Specifying "*" for <name> deletes all.
length Number of fields, or the number of characters in a string or symbol
nth$ Specified field if exists, else nil
member$ Number of the field if literal or variable exists, else FALSE
subsetp Returns TRUE if a multifield value is a subset of another multifield value, else FALSE
delete$ Given a field number, deletes the value in the field
explode$ Each string element is returned as part of a new multifield value
subseq$ Returns a specified range of fields
replace$ Replaces a specified value

3.4 More about functions

A function returns only the last action, the other are not, with the exception of the printout function which can be placed anywhere, not only as the last action. Also, at the beginning of a function description we can include a list of ṭdummyṭ arguments, in fact the local names of the parameters that the function gets when it is called.
The syntax of the deffunction is as follows:

```
(deffunction <function-name> [optional comment]
  (?arg1 ?arg2 ...?argM [$?argN]) ;argument list. Last one may
  (<action1>         ;be optional multifield arg.
  <action2>          ;action1 to
  ...                ;action(K-1) do not
<action(K-1)>        ;return a value
<actionK>)           ;only last action returned
```

An example:

```
CLIPS> (clear)
CLIPS> (deffunction hypotenuse ; name
          (?a ?b) ; dummy arguments
          (sqrt (+ (* ?a ?a) (* ?b ?b)))) ; action
CLIPS> (defrule calculate-hypotenuse
          (dimensions ?base ?height)
          =>
          (printout t "Hypotenuse=" (hypotenuse ?base ?height)
     crlf))
```

```
CLIPS> (assert (dimensions 3 4))
<Fact-0>
CLIPS> (run)
Hypotenuse=5.0
CLIPS>
```

Reference ug.pdf pages 70-73

3.5 How to write a C/C++ function that I can call in CLIPS

This thing can be used for example for the purpose of writing a probing function for a sensor that collects data from the real world and can be called from a CLIPS "program". The example has the purpose of describing how to write a simple function not such a complex one as suggested above. It is described in [apg.pdf, pages 19-42] especially in the last paragraph [apg.pdf, pages 39-42] where a complete example and a step-by-step way to accomplish the task is described. Mainly it assumes writing the function in an external file, modifying then one of the source files from CLIPS, then recompiling all the sources from CLIPS, including the newly created user-written files containing the new function(s). Thus recompiling CLIPS.
I will to detail this step (of recompilation) using Visual Studio. The variant of compiling it under Unix/Linux seems to be preferred on the CLIPS website on SourceForge (you can find the make files, but the projects fro Visual Studio don't maintain the name and structure of the folders contained… you have to move the files arround for the thing to work).

For now, only the steps from [apg.pdf, pages 39-42].

1) Copy all of the CLIPS source code file to the user directory.
2) Define the user function in a new file.

```
#include "clips.h"
double TripleNumber()
{
return(3.0 * RtnDouble(1));
}
```

The preceding function does the job just fine. The following function, however, accomplishes the
same purpose while providing error handling on arguments and allowing either an integer or double return value.

```
#include "clips.h"
void TripleNumber(
DATA_OBJECT_PTR returnValuePtr)
{
void *value;
long longValue;
double doubleValue;
/*================================================*/
/* If illegal arguments are passed, return zero. */
/*================================================*/
if (ArgCountCheck("triple",EXACTLY,1) == -1)
{
SetpType(returnValuePtr,INTEGER);
SetpValue(returnValuePtr,AddLong(0L));
return;
```

```
}
if (! ArgTypeCheck("triple",1,INTEGER_OR_FLOAT,returnValuePtr))
{
SetpType(returnValuePtr,INTEGER);
SetpValue(returnValuePtr,AddLong(0L));
return;
}
/*====================*/
/* Triple the number. */
/*====================*/
if (GetpType(returnValuePtr) == INTEGER)
{
value = GetpValue(returnValuePtr);
longValue = 3 * ValueToLong(value);
SetpValue(returnValuePtr,AddLong(longValue));
}
else /* the type must be FLOAT */
{
value = GetpValue(returnValuePtr);
doubleValue = 3.0 * ValueToDouble(value);
SetpValue(returnValuePtr,AddDouble(doubleValue));
}
return;
}
```

3) Define the constructs which use the new function in a new file (or in an existing constructs
file). For example:

```
(deffacts init-data
(data 34)
(data 13.2))
(defrule get-data
(data ?num)
=>
(printout t "Tripling " ?num crlf)
(assert (new-value (triple ?num))))    ; this is the call to the user-defined
function
(defrule get-new-value
(new-value ?num)
=>
(printout t crlf "Now equal to " ?num crlf))
```

4) Modify the CLIPS **userfunctions.c** file to include the new UserFunctions definition.

```
void UserFunctions()
{
/* The following code is used with the second example */
/* of the TripleFunction listed in step 2. */
```



```
extern void TripleNumber(DATA_OBJECT_PTR);
DefineFunction2("triple",'u',PTIF TripleNumber, "TripleNumber",
"11n");
/* Alternately, if the TripleFunction with a double return */
/* value from step 2 was used, the following declaration */
/* and DefineFunction2 call should be used in place of the */
/* one above. */
/*
extern double TripleNumber(void);
DefineFunction2("triple",'d',PTIF TripleNumber, "TripleNumber","11n");
*/
}
```

5) Compile the CLIPS files along with any files which contain user-defined functions.
6) Link all object code files.
7) Execute new CLIPS executable. Load the constructs file and test the new function.

You can notice that at step 3, the new function call is (triple ?num), we use the name defined at step 4 with the definition

```
extern double TripleNumber(DATA_OBJECT_PTR);
DefineFunction2("triple",'u',PTIF TripleNumber, "TripleNumber", "11n");
```

That is the sequence for defining the external function that returns a double (the short definition from step 2). TripleNumber is the name of the function as defined in the C code and triple is the name that will be used in CLIPS. More about what means "11n" in section 3.6. To be more precise, there you can learn what "u" does, and for the fifth argument of the **DefineFunction2**, which is exactly "11n" you will be sent to the external reference [apg.pdf page 22(42)].
Also, on step 5 and 6 you need to compile and link all the CLIPS sources. Read further or **go to section 3.6** where this is explained. Return and read section 3.5.1 for parameters in the userfunctions.c.

In order to do that you need to download the CLIPS sources from the CLIPS website, that is, for version 6.2.4

http://sourceforge.net/projects/clipsrules/files/CLIPS/6.24/windows_ide_source_624.zip/download
and
http://sourceforge.net/projects/clipsrules/files/CLIPS/6.24/clips_core_source_624.zip/download

Then you have to unpack them and, with your files as described at steps 2), and 4) compile the project with Visual Studio and obtain another version of the CLIPS executable, which includes also your new function(s). In the folder pc-prjct\CLIPS\MVC\ from the windows_ide_source_624.zip file you already have a Visual Studio project file, but the folders might be slightly different so you probably have to look at the errors and move the files around in order to work. Or create your own project. I personally think that creating 2 folders in the project and copying the files into them is easier.
Second version for compiling the sources: follow the step described in 3.6.

3.5.1 Parameters for userfunctions.c

In the userfunction.c file mentioned at step 4), the significance of the DefineFunction (and DefineFunction2) parameters is the following:

a) The first argument is the CLIPS function name, a string that will be used in CLIPS when calling the function from within CLIPS
b) The second argument is the return type of the function, that is the type of the value that will be returned to CLIPS. Allowable return types are:

Return Code	Return Type Expected
a	External Address
b	Boolean
c	Character
d	Double Precision Float
f	Single Precision Float
i	Integer
j	Unknown Data Type (Symbol, String, or Instance Name Expected)
k	Unknown Data Type (Symbol or String Expected)

l	Long Integer
m	Multifield
n	Unknown Data Type (Integer or Float Expected)
o	Instance Name
s	String
u	Unknown Data Type (Any Type Expected)
v	Void—No Return Value
w	Symbol
x	Instance Address

c) The fifth argument from DefineFunction2 (DefineFunction only has 4 arguments) is a restriction string which indicates the number and types of the arguments that CLIPS function expects. The syntax for the restriction string is:

```
<min-args> <max-args> [<default-type> <types>*]
```

d) More information about that in [apg.pdf page 22(42)] where you can see also some examples for the fifth argument

3.6 How to compile and build the CLIPS Executables with Microsoft Visual Studio

Details about that topic can be found in [ig.pdf, page 33(49)] and can be resumed to the following steps:

1) Download the sources for the Windows interface, and CLIPS from the website. For version 6.2.4, the links are:
http://sourceforge.net/projects/clipsrules/files/CLIPS/6.24/windows_ide_source_624.zip/download
and
http://sourceforge.net/projects/clipsrules/files/CLIPS/6.24/clips_core_source_624.zip/download
2) When you unpack the windows interface, you obtain some folders, in one of them you can find the files **CLIPSwin.sln**, **CLIPSWin.vcproj** and **CLIPS.vcproj**. Create a folder and place the 3 files in it. It will be the project's files.
3) Create a subdirectory named Source in the project's directory. Create a subdirectory named CLIPS in the Source directory. Copy all the source files into the CLIPS directory. Those are obtained after extracting them from the second archive (clips_core_source_624.zip)
4) Create a subdirectory named Interface in the Source directory. Copy the Windows interface source code in this folder. The files that you will need to copy are in the first archive and are already there in a folder called Interface.
5) Launch the Microsoft Visual C++ by double clicking on the **CLIPSwin.sln** file.

3.7 How to embed CLIPS into other programs

A complete guide of the functions that you can use can be found in apg.pdf, starting from page 43(63). Mainly for the use of CLIPS in C/C++ programs you need to include the following
#include <stdio.h>
#include "clips.h"
The second one, clips.h uses, at his turn his own includes, so, probably you need to attach to your project the entire CLIPS sources, downloadable from the following link (for version 6.2.4)
http://sourceforge.net/projects/clipsrules/files/CLIPS/6.24/clips_core_source_624.zip/download

"The user's main program must initialize CLIPS by calling the function **InitializeEnvironment** at some time prior to loading constructs. **UserFunctions** and

EnvUserFunctions also must be defined, regardless of whether CLIPS calls any external functions. Compile and link all of the user's code with all CLIPS files *except* the object version of **main.c**. When running CLIPS as an embedded program, many of the capabilities available in the interactive interface (in addition to others) are available through function calls" (apg.pdf, page 43(63)

So, the steps for writing a program which embeds CLIPS, as described in [apg.pdf, page 148(168)] are presented below. Note that the steps do almost the same as the previous example, bu now as an embedded application.

1) Copy all of the CLIPS source code file to the user directory.
2) Define the user function (TripleNumber) and a new main routine in a new file. These could
go in separate files if desired. For this example, they will all be included in a single file.

```
#include "clips.h"
main()
{
InitializeEnvironment();
Load("constructs.clp");
Reset();
Run(-1L)
}
void TripleNumber(
DATA_OBJECT_PTR returnValuePtr)
{
void *value;
long longValue;
double doubleValue;
/*================================================*/
/* If illegal arguments are passed, return zero. */
/*================================================*/
if (ArgCountCheck("triple",EXACTLY,1) == -1)
{
SetpType(returnValuePtr,INTEGER);
SetpValue(returnValuePtr,AddLong(0L));
return;
}
if (! ArgTypeCheck("triple",1,INTEGER_OR_FLOAT,returnValuePtr))
{
SetpType(returnValuePtr,INTEGER);
SetpValue(returnValuePtr,AddLong(0L));
return;
}
/* Triple the number. */
/*====================*/
if (GetpType(returnValuePtr) == INTEGER)
{
value = GetpValue(returnValuePtr);
longValue = 3 * ValueToLong(value);
SetpValue(returnValuePtr,AddLong(longValue));
}
else /* the type must be FLOAT */
{
value = GetpValue(returnValuePtr);
doubleValue = 3.0 * ValueToDouble(value);
SetpValue(returnValuePtr,AddDouble(doubleValue));
}
```

```
return;
}
```

3) Modify UserFunctions in the CLIPS **userfunctions.c** file.

```
void UserFunctions()
{
extern void TripleNumber();
DefineFunction2("triple",'u',PTIF TripleNumber, "TripleNumber",
"11n");
}
void EnvUserFunctions(
void *theEnv))
{
}
```

4) Define constructs which use the new function in a file called **constructs.clp** (or any file; just
be sure the call to **Load** loads all necessary constructs prior to execution).

```
(deffacts init-data
(data 34)
(data 13.2))
(defrule get-data
(data ?num)
=>
(printout t "Tripling " ?num crlf)
(assert (new-value (triple ?num))))
(defrule get-new-value
(new-value ?num)
=>
(printout t crlf "Now equal to " ?num crlf))
```

5) Compile all CLIPS files, *except* **main.c**, along with all user files.
6) Link all object code files.
7) Execute new CLIPS executable.

3.8 How to use deffunctions and generic functions and manipulate CLIPS objects from C

This section lists the steps needed to define and use an embedded CLIPS application. The example illustrates how to call deffunctions and generic functions as well as manipulate objects from C. [apg.pdf, page

1) Copy all of the CLIPS source code file to the user directory.
2) Define a new main routine in a new file.

```
#include <stdio.h>
#include "clips.h"
main()
{
void *c1,*c2,*c3;
DATA_OBJECT insdata,result;
char numbuf[20];
InitializeEnvironment();
/*=========================================================*/
/* Load the classes, message-handlers, generic functions */
/* and generic functions necessary for handling complex */
/* numbers. */
/*=========================================================*/
Load("complex.clp");
```

```
/*=========================================================*/
/* Create two complex numbers. Message-passing is used to */
/* create the first instance c1, but c2 is created and has */
/* its slots set directly. */
/*=========================================================*/
c1 = MakeInstance("(c1 of COMPLEX (real 1) (imag 10))");
c2 = CreateRawInstance(FindDefclass("COMPLEX"),"c2");
result.type = INTEGER;
result.value = AddLong(3L);
DirectPutSlot(c2,"real",&result);
result.type = INTEGER;
result.value = AddLong(-7L);
DirectPutSlot(c2,"imag",&result);
/*=========================================================*/
/* Call the function '+' which has been overloaded to handle */
/* complex numbers. The result of the complex addition is */
/* stored in a new instance of the COMPLEX class. */
/*=========================================================*/
FunctionCall("+","[c1] [c2]",&result);
c3 = FindInstance(NULL,DOToString(result),TRUE);
/*=========================================================*/
/* Print out a summary of the complex addition using the */
/* "print" and "magnitude" messages to get information */
/* about the three complex numbers. */
/*=========================================================*/
PrintRouter("stdout","The addition of\n\n");
SetType(insdata,INSTANCE_ADDRESS);
SetValue(insdata,c1);
Send(&insdata,"print",NULL,&result);
PrintRouter("stdout","\nand\n\n");
SetType(insdata,INSTANCE_ADDRESS);
SetValue(insdata,c2);
Send(&insdata,"print",NULL,&result);
PrintRouter("stdout","\nis\n\n");
SetType(insdata,INSTANCE_ADDRESS);
SetValue(insdata,c3);
Send(&insdata,"print",NULL,&result);
PrintRouter("stdout","\nand the resulting magnitude is\n\n");
SetType(insdata,INSTANCE_ADDRESS);
SetValue(insdata,c3);
Send(&insdata,"magnitude",NULL,&result);
sprintf(numbuf,"%lf\n",DOToDouble(result));
PrintRouter("stdout",numbuf);
}
```

3) Define constructs which use the new function in a file called **complex.clp** (or any file; just be sure the call to **Load** loads all necessary constructs prior to execution).

```
(defclass COMPLEX (is-a USER)
(role concrete)
(slot real (create-accessor read-write))
(slot imag (create-accessor read-write)))
(defmethod + ((?a COMPLEX) (?b COMPLEX))
(make-instance of COMPLEX
(real (+ (send ?a get-real) (send ?b get-real)))
(imag (+ (send ?a get-imag) (send ?b get-imag)))))
(defmessage-handler COMPLEX magnitude ()
(sqrt (+ (** ?self:real 2) (** ?self:imag 2))))
```

4) Compile all CLIPS files, *except* **main.c**, along with all user files.

5) Link all object code files.

6) Execute new CLIPS executable.

3.9 How to use CLIPS Java Native Interface (CLIPSJNI)

The download for CLIPSJNI exists on the CLIPS website only in the last (beta) version's folder. The link is:
http://sourceforge.net/projects/clipsrules/files/CLIPS/6.30/CLIPSJNI_0.3.zip/download

If you download the file, unzip it, you have, the folder obtained, somewhere a file called instructions.pdf. It says there how to run the examples contained from the command line. Of course, since it uses java you need to have java installed (you probably already have it, it is sufficient to have the JRE).
On page 4(193) you can see, for example, that, for running the Sudoku example, you should:

a) Open a Command prompt
b) Go to the CLIPSJNI\examples\SudokuDemo folder, using the cd command. For example, if you have unpacked the CLIPSJNI folder in the CLIPS folder, which is on drive c: you could change the directory with the command cd c:\CLIPS\CLPISJNI\examples\SudokuDemo
c) Launch the demo with the command
```
java -cp .;../../CLIPSJNI.jar -Djava.library.path=../.. SudokuDemo
```
Notice the classpath to be the current directory (.) and the path to the CLIPSJNI.jar file. Of course if you launch the command from another folder, then both this elements will have to be modified accordingly with the new location
d) Your demo, with a graphical interface is ready to be used… if you know Sudoku, of course. If not you can choose another demo, for example the WineDemo exemple, which is easier to use. Of course, on steps b) and c) you will choose the WineDemo folder, respectively the WineDemo class, instead fo SudokuDemo

On the following website you can find examples of how to use CLIPSJNI through java servlets thus obtaining web applications.
http://www.csie.ntu.edu.tw/~sylee/courses/clips/clipsjni.htm