CLIPS 6.4
User's Guide

by Joseph C. Giarratano, Ph.D.

Table of Contents

Readme

The first step on the road to wisdom is the admission of ignorance. The second step is realizing that you don't have to blab it to the world.

This section was formerly called the *Preface*, but since nobody read it, I renamed it to a more conventional title that computers users are conditioned to obey. Another suggestion was to call this the *Don't Readme* section, but since people today believe everything they read, I was afraid they really wouldn't read it.

The purpose of a *Preface*, oops, excuse me, a *Readme*, is to provide **metaknowledge** about the knowledge contained in a book. The term *metaknowledge* means knowledge about the knowledge. So this description of the *Readme* is actually metametaknowledge. If you're either confused or intrigued at this point, go ahead and read this book anyway because I need all the readers I can get.

What Is CLIPS?

CLIPS is an expert system tool originally developed by the Software Technology Branch (STB), NASA/Lyndon B. Johnson Space Center. Since its first release in 1986, CLIPS has undergone continual refinement and improvement. It is now used by thousands of people around the world.

CLIPS is designed to facilitate the development of software to model human knowledge or expertise.

There are three ways to represent knowledge in CLIPS:

- *Rules*, which are primarily intended for heuristic knowledge based on experience.

- *Deffunctions* and *generic functions*, which are primarily intended for procedural knowledge.

- *Object-oriented programming*, also primarily intended for procedural knowledge. The five generally accepted features of object-oriented programming are supported: classes, message-handlers, abstraction, encapsulation, inheritance, and polymorphism. Rules may pattern match on objects and facts.

You can develop software using only rules, only objects, or a mixture of objects and rules.

CLIPS has also been designed for integration with other languages such as C and Java. In fact, CLIPS is an acronym for C Language Integrated Production System. Rules and objects form an integrated system too since rules can pattern-match on facts and objects. In addition to being used as a stand-alone tool, CLIPS can be called from a procedural language, perform its function, and then return control back to the calling program. Likewise, procedural code can be defined as external functions and called from CLIPS. When the external code completes execution, control returns to CLIPS.

If you are already familiar with object-oriented programming in other languages such as Smalltalk, C++, Objective C, or Java, you know the advantages of objects in developing software. If you are not familiar with object-oriented programming, you will find that CLIPS is an excellent tool for learning this new concept in software development.

What This Book is About

The *CLIPS User's Guide* is an introductory tutorial on the basic features of CLIPS. It is *not* intended to be a comprehensive discussion of the entire tool. The companion volume to this book is the *CLIPS Reference Manual*, which does provide a complete, comprehensive discussion of all the topics in this book and much more.

Who Should Read This Book

The purpose of the *CLIPS User's Guide* is to provide an easy to read, elementary introduction to expert systems for people with little or no experience with expert systems.

The *CLIPS User's Guide* can be used in the classroom or for self-teaching. The only prerequisite is that you have a basic knowledge of programming in a high-level language such as Java, Ada, FORTRAN, C (OK, BASIC if nothing else, but we won't admit it in public and will disavow this statement if asked.)

How To Use This Book

The CLIPS User's Guide is designed for people who want a quick introduction to expert system programming in a hands-on manner. The examples are of a very general nature. Also, since learning a new language can be a frustrating experience, the writing is in a light, humorous style (I hope) compared to serious-minded, massive, and intimidating college textbooks. Hopefully, the humor will not offend anyone with a sense of humor.

For maximum benefit, you should type in the example programs in the text as you read through the book. By typing in the examples, you will see how the programs should work and what error messages occur if you make a mistake. The output for the examples is shown or described after each example. Finally, you should read the corresponding material in the CLIPS Reference Manual as you cover each chapter in the CLIPS User's Guide.

Like any other programming language, you will only learn programming in CLIPS by writing programs in it. To really learn expert system programming, you should pick a problem of interest and write it in CLIPS.

Acknowledgments

I greatly appreciate the advice and reviews of this book by many people. Thanks to Gary Riley, Chris Culbert, Brian Dantes, Bryan Dulock, Steven Lewis, Ann Baker, Steve Mueller, Stephen Baudendistel, Yen Huynh, Ted Leibfried, Robert Allen, Jim Wescott, Marsha Renals, Pratibha Boloor, Terry Feagin, and Jack Aldridge. Special thanks to Bob Savely for supporting the development of CLIPS.

Editor's Note

Minor changes have been made to Dr. Giarratano's original document to reflect changes in newer versions of CLIPS — Gary Riley.

Chapter 1

Just the Facts

If you ignore the facts, you'll never worry about being wrong

This chapter introduces the basic concepts of an expert system. You'll see how to insert and remove facts in CLIPS.

Introduction

CLIPS is a type of computer language designed for writing applications called **expert systems**. An expert system is a program which is specifically intended to model human expertise or knowledge. In contrast, common programs such as payroll programs, word processors, spreadsheets, computer games, and so forth, are not intended to embody human expertise or knowledge. (One definition of an expert is someone more than 50 miles from home and carrying a briefcase.)

CLIPS is called an expert system **tool** because it is a complete environment for developing expert systems which includes features such as an integrated editor and a debugging tool. The word **shell** is reserved for that portion of CLIPS which performs **inferences** or reasoning. The CLIPS shell provides the basic elements of an expert system:

1. **fact-list**, and **instance-list:** Global memory for data

2. **knowledge-base**: Contains all the rules, the **rule-base**

3. **inference engine**: Controls overall execution of rules

A program written in CLIPS may consist of rules, facts, and objects. The inference engine decides which rules should be executed and when. A rule-based expert system written in CLIPS is a data-driven program where the facts, and objects if desired, are the data that stimulate execution via the inference engine.

This is one example of how CLIPS differs from procedural languages such as Java, Ada, BASIC, FORTRAN, and C. In procedural languages, execution can proceed without data. That is, the statements are sufficient in those languages to cause execution. For example, a statement such as PRINT 2 + 2 could be immediately executed in BASIC. This is a complete statement

that does not require any additional data to cause its execution. However, in CLIPS, data are required to cause the execution of rules.

Originally, CLIPS had capabilities to represent only rules and facts. However, the enhancements of Version 6.0 allow rules to match objects as well as facts. Also, objects may be used *without* rules by sending messages and so the inference engine is no longer necessary if you use only objects. In chapters 1 through 7, we'll discuss the facts and rules of CLIPS. The object features of CLIPS are covered in chapters 8 through 12.

The Beginning and the End

To begin CLIPS, just enter the appropriate run command for your system. You should see the CLIPS prompt appear as follows:

```
CLIPS>
```

At this point, you can start entering **commands** directly into CLIPS. The mode in which you are entering direct commands is called the **top-level**. If you have a Graphical User Interface (GUI) version of CLIPS, you can also **select** some commands using the mouse or arrow keys rather than typing them in. Please refer to the *CLIPS Interfaces Guide* for a discussion of the commands supported by the various CLIPS GUIs. For simplicity and uniformity in this book, we'll assume the commands are typed in.

The normal mode of leaving CLIPS is with the **exit** command. Just type

```
(exit)
```

in response to the CLIPS prompt and then press the carriage return key.

Making a List

As with other programming languages, CLIPS recognizes certain keywords. For example, if you want to put data in the fact-list, you can use the assert command.

As an example of *assert*, enter the following right after the CLIPS prompt as shown:

```
CLIPS> (assert (duck))
```

Here the assert command takes (duck) as its argument. Be sure to always press the carriage return key to send the line to CLIPS.

You will see the response

```
<Fact-1>
```

which indicates CLIPS has stored the duck fact in the fact-list and given it the identifier 1. The **angle-brackets** are used as a delimiter in CLIPS to surround the name of an item. CLIPS will automatically name facts using a sequentially increasing number and list the highest fact-index when one or more facts is asserted.

Notice that the (assert) command and its (duck) argument are surrounded by parentheses. Like many other expert system languages, CLIPS has a LISP-like syntax which uses parentheses as delimiters. Although CLIPS is not written in LISP, the style of LISP has influenced the development of CLIPS.

And Checking It Twice

Suppose you want to see what's in the fact-list. If your version of CLIPS supports a GUI, you may just select the appropriate command from the menu. Alternatively, you can enter commands from the keyboard. In the following, we'll describe the keyboard commands since the window selections are self-explanatory.

The keyboard command to see facts is with the **facts command**. Enter (facts) in response to the CLIPS prompt and CLIPS will respond with a list of facts in the fact-list. Be sure to put parentheses around the command or CLIPS will not accept it. The result of the (facts) command in this example should be

```
CLIPS> (facts)
f-1     (duck)
For a total of 1 fact.
CLIPS>
```

The term *f-1* is the **fact identifier** assigned to the (duck) fact by CLIPS. Every fact inserted into the fact-list is assigned a unique fact identifier starting with the letter "f" and followed by an integer called the **fact-index**. On starting up CLIPS, and after certain commands such as **clear** and **reset** (to be discussed in more detail later), the fact- index will be set to one, and then incremented by one as each new fact is asserted.

What happens if you try to put a second duck into the fact-list? Let's try it and see. Assert a new (duck), then issue a (facts) command as follows

```
CLIPS> (assert (duck))
<Fact-1>
CLIPS> (facts)
```

```
f-1      (duck)
For a total of 1 fact.
CLIPS>
```

The <Fact-1> message is returned by CLIPS to indicate that the fact already exists. You'll see just the original "f-1 (duck)". This shows that CLIPS will not accept a duplicate entry of a fact. However, there is an override command, **set-fact-duplication**, which will allow duplicate fact entry.

Of course you can put in other, different facts. For example, assert a (quack) fact and then issue a (facts) command. You'll see

```
CLIPS> (assert (quack))
<Fact-2>
CLIPS> (facts)
f-1      (duck)
f-2      (quack)
For a total of 2 facts.
CLIPS>
```

Notice that the (quack) fact is now in the fact-list.

Facts may be removed or **retracted**. When a fact is retracted, the other facts do not have their indices changed, and so there may be "missing" fact-indices. As an analogy, when a football player leaves a team and is not replaced, the jersey numbers of the other players are not all adjusted because of the missing number (unless they really hate the guy's guts and want to forget he ever played for them).

Clearing Up the Facts

The **clear** command removes all facts from memory, as shown by the following.

```
CLIPS> (facts)
f-1      (duck)
f-2      (quack)
For a total of 2 facts.
CLIPS> (clear)
CLIPS> (facts)
CLIPS>
```

The (clear) command essentially restores CLIPS to its original startup state. It clears the memory of CLIPS and resets the fact-identifier to one. To see this, assert (animal-is duck), then check the fact-list.

```
CLIPS> (assert (animal-is duck))
<Fact-1>
CLIPS> (facts)
f-1     (animal-is duck)
For a total of 1 fact.
CLIPS>
```

Notice that (animal-is duck) has a fact-identifier of f-1 because the (clear) command reset the fact identifiers. The (clear) command actually does more than just remove facts. Besides removing all the facts, (clear) also removes all the rules, as you'll see in the next chapter.

Sensitive Fields and Slurping

A fact such as (duck) or (quack) is said to consist of a single **field**. A *field* is a placeholder (named or unnamed) that may have a value associated with it. As a simple analogy, you can think of a field as a picture frame. The frame can hold a picture, perhaps a picture of your pet duck (For those of you who are curious what a picture of a "quack" looks like, it could be (1) a photo of an oscilloscope trace of a duck saying "quack", where the signal input comes from a microphone, or (2) for those of you who are more scientifically inclined, a Fast Fourier Transform of the "quack" signal, or (3) a TV-huckster selling a miracle cure for wrinkles, losing weight, etc.). Named placeholders are only used with **deftemplates**, described in more detail in chapter 5.

The (duck) fact has a single, unnamed placeholder for the value duck. This is an example of a **single-field** fact. A field is a placeholder for a value. As an analogy to fields, think of dishes (fields) for holding food (values).

The **order** of unnamed fields is significant. For example, if a fact was defined

```
(Brian duck)
```

and interpreted by a rule as the hunter Brian shot a duck, then the fact

```
(duck Brian)
```

would mean that the hunter duck shot a Brian. In contrast, the order of named fields is not significant, as you'll see later with deftemplate.

Actually, it is good software engineering to start the fact with a relation that describes the fields. A better fact would be

```
(hunter-game duck Brian)
```

to imply that the first field is the hunter and the second field is the game.

A few definitions are now necessary. A **list** is a group of items with no implied order. Saying that a list is **ordered** means that the position in the list is significant. A **multifield** is a sequence of fields, each of which may have a value. The examples of (duck Brian) and (Brian duck) are multifield facts. If a field has no value, the special symbol **nil**, which means "nothing" may be used for an empty field as a placeholder. For example,

```
(duck nil)
```

would mean that the killer duck bagged no trophies today.

Note that the *nil* is necessary to indicate a placeholder, even if it has no value. For example, think of a field as analogous to a mailbox. There's a big difference between an empty mailbox, and no mailbox at all. Without the *nil*, the fact becomes a single-field fact (duck). If a rule depends on two fields, it will not work with only one field, as you'll see later.

There are a number of different **types** of fields available: **float**, **integer**, **symbol**, **string**, **external-address**, **fact-address**, **instance-name** and **instance-address** The type of each field is determined by the type of **value** stored in the field. In an unnamed field, the type is determined *implicitly* by what type you put in the field. In deftemplates, you can *explicitly* declare the type of value that a field can contain. The use of explicit types enforces the concepts of **software engineering**, which is a discipline of programming to produce quality software.

A **symbol** is one type of field that starts with a printable ASCII character and is followed optionally by zero or more printable characters. Fields are commonly **delimited** or bounded, by one or more spaces or parentheses. For example,

```
(duck-shot Brian Gary Rey)
```

However, this could be a legal deftemplate fact if "shot" is defined as the name of a field, while "Brian Gary Rey" are the values associated with the named field.

CLIPS is case-sensitive. Also, certain characters have special meaning to CLIPS.

```
"   (   )   &   |   <   ~ ;   ?   $
```

The "&", "|", and "~" may not be used as stand-alone symbols or as *any* part of a symbol.

Some characters act as delimiters by *ending* a symbol. The following characters act as delimiters for symbols.

- any non-printable ASCII character, including spaces, carriage returns, tabs, and linefeeds

- double quotes, **"**

- opening and closing parentheses, **()**

- ampersand, **&**

- vertical bar, **|**

- less than, **<**. Note that this may be the *first* character of a symbol

- tilde, **~**

- semicolon, **;** indicates start of a comment, a carriage return ends it

- ? and $? may not begin a symbol but may be inside it

The semicolon acts as the start of a comment in CLIPS. If you try to assert a semicolon, CLIPS will think you're entering a comment and wait for you to finish. If you accidentally enter a semicolon in top-level, just type in a closing parenthesis and carriage return. CLIPS will respond with an error message and the CLIPS prompt will reappear (This is one of the few approved occasions in life in which it's necessary to do something wrong to get something right.)

As you read through this manual, you will learn the special meanings of the characters above. With the exception of the "&", "|", and "~", you may use the others as described. However, it may be confusing to someone reading your program and trying to understand what the program is doing. In general, it's best to avoid using these characters in symbols unless you have some good reason for using them.

The following are examples of symbols.

```
duck
duck1
duck_soup
duck-soup
duck1-1_soup-soup
d!?#%^
```

The second type of field is the **string**. A string must begin and end with double quotes. The double quotes are part of the field. Zero or more characters of any kind can appear between the double quotes. Some examples of strings follow.

```
"duck"
"duck1"
"duck/soup"
"duck soup"
"duck soup is good!!!"
```

The third and fourth types of field are **numeric fields**. A field which represents a number can be either an **integer** or **floating-point** type field. A floating-point type is commonly referred to simply as a **float.**

All numbers in CLIPS are treated as "long long" integers or **double-precision** floats. Numbers without a decimal point are treated as integers unless they are outside integer range. The range is machine dependent on the number of bits, N, used to represent the integer as follows.

```
 - 2^(N-1)

        .

        .

        .

  2^(N-1) - 1
```

For 64-bit "long long" integers, this corresponds to a range of numbers

```
 -  9,223,372,036,854,775,808

         .

         .

         .

    9,223,372,036,854,775,807
```

As some examples of numbers, assert the following data where the last number is in exponential notation, and uses the "e" or "E" for the power-of-ten.

```
CLIPS> (clear)
CLIPS> (facts)
CLIPS> (assert (number 1))
<Fact-1>
```

```
CLIPS> (assert (x 1.5))
<Fact-2>
CLIPS> (assert (y -1))
<Fact-3>
CLIPS> (assert (z 65))
<Fact-4>
CLIPS> (assert (distance 3.5e5))
<Fact-5>
CLIPS> (assert (coordinates 1 2 3))
<Fact-6>
CLIPS> (assert (coordinates 1 3 2))
<Fact-7>
CLIPS> (facts)
f-1      (number 1)
f-2      (x 1.5)
f-3      (y -1)
f-4      (z 65)
f-5      (distance 350000.0)
f-6      (coordinates 1 2 3)
f-7      (coordinates 1 3 2)
For a total of 7 facts.
CLIPS>
```

As you can see, CLIPS prints the number entered in exponential notation as 350000.0 because it converts from power-of-ten format to floating-point if the number is small enough.

Notice that each fact must start with a symbol such as "number", "x", "y", etc. Before CLIPS version 6.0, it was possible to enter only a number as a fact. However, now a symbol is required as the first field. Also, certain reserved words used by CLIPS cannot be used as the first field, but may be used for other fields. For example, the reserved word *not* is used to indicate a negated pattern and may not be used as the first field of a fact.

A **fact** consists of one or more fields enclosed in matching left and right parentheses. For simplicity we'll only discuss facts in the first seven chapters, but most of the discussion of pattern matching applies to objects as well. Exceptions are certain functions such as assert and retract which only apply to facts, not objects. The corresponding ways to handle objects are discussed in chapters 8–12.

A fact may be **ordered** or **unordered**. All the examples you've seen so far are ordered facts because the order of fields makes a difference. For example, notice that CLIPS considers these as separate facts although the same values "1", "2", and "3" are used in each.

```
f-6     (coordinates 1 2 3)
f-7     (coordinates 1 3 2)
```

Ordered facts *must* use field position to define data. As an example, the ordered fact (duck Brian) has two fields and so does (Brian duck). However, these are considered as two separate facts by CLIPS because the order of field values is different. In contrast, the fact (duck-Brian) has only one field because of the "-" concatenating the two values.

Deftemplate facts, described in more detail later, are unordered because they use named fields to define data. This is analogous to the use of structs in C and other languages.

Multiple fields normally are separated by **white space** consisting of one or more spaces, tabs, carriage returns, or linefeeds. For example, enter the following examples as shown and you'll see that each stored fact is the same.

```
CLIPS> (clear)
CLIPS> (assert (The duck says "Quack"))
<Fact-1>
CLIPS> (facts)
f-1     (The duck says "Quack")
For a total of 1 fact.
CLIPS> (clear)
CLIPS> (assert  (The     duck     says     "Quack"  ))
<Fact-1>
CLIPS> (facts)
f-1     (The duck says "Quack")
For a total of 1 fact.
CLIPS>
```

Carriage returns may also be used to improve readability. In the following example, a carriage return is typed after every field and the asserted fact is the same as before when the fact was entered on one line.

```
CLIPS> (clear)
CLIPS> (assert (The
duck
says
```

```
    "Quack"))
    <Fact-1>
    CLIPS> (facts)
    f-1     (The duck says "Quack")
    For a total of 1 fact.
    CLIPS>
```

However, be careful if you insert a carriage return inside of a string, as the following example shows.

```
    CLIPS> (assert (The
    duck
    says
    "Quack
    "))
    <Fact-2>
    CLIPS> (facts)
    f-1     (The duck says "Quack")
    f-2     (The duck says "Quack
    ")
    For a total of 2 facts.
    CLIPS>
```

As you can see, the carriage return embedded in the double quotes was output with the string to put the closing double quote on the next line. This is important because CLIPS considers fact f-1 as distinct from fact f-2.

Notice also that CLIPS preserved the uppercase and lowercase letters in the fields of the fact. That is, the "T" of "The" and the "Q" of "Quack" are uppercase. CLIPS is said to be **case-sensitive** because it distinguishes between uppercase and lowercase letters. For example, assert the facts (duck) and (Duck) and then issue a (facts) command. You'll see that CLIPS allows you to assert (duck) and (Duck) as different facts because CLIPS is case-sensitive.

The following example is a more realistic case in which carriage returns are used to improve the readability of a list. To see this, assert the following fact where carriage returns and spaces are used to put fields at appropriate places on different lines. Dashes or minus signs are used intentionally to create single fields, so CLIPS will treat items like "fudge sauce" as a single field.

```
    CLIPS> (clear)
    CLIPS>
    (assert (grocery-list
```

```
                ice-cream
                cookies
                candy
                fudge-sauce))
   <Fact-1>
   CLIPS> (facts)
   f-1     (grocery-list ice-cream cookies candy fudge-sauce)
   For a total of 1 fact.
   CLIPS>
```

As you can see, CLIPS replaced the carriage returns and tabs with single spaces. While the use of white space in separating the facts is convenient for a person reading a program, they are converted to single spaces by CLIPS.

A Matter of Style

It is good rule-based programming style to use the first field of a fact to describe the *relationship* of the following fields. When used this way, the first field is called a *relation*. The remaining fields of the fact are used for specific values. An example is (grocery-list ice-cream cookies candy fudge-sauce). The dashes are used to make multiple words fit in a single field.

Good documentation is even more important in an expert system than in languages such as Java, C, Ada, etc., because the rules of an expert system are not generally executed in a sequential manner. CLIPS aids the programmer in writing descriptive facts like this by means of deftemplates.

Another example of related facts is (duck), (horse), and (cow). It's better style to refer to them as

```
   (animal-is duck)
   (animal-is horse)
   (animal-is cow)
```

or as the single fact

```
   (animals duck horse cow)
```

since the relation *animal-is* or *animals* describes their relation and so provides some documentation to the person reading the code.

The explicit relations, *animal-is* and *animals*, make more sense to a person than the implicit meaning of (duck), (horse), and (cow). While this example is simple enough that anyone can figure out the implicit relations, it is an easy trap to fall into to write facts in which the relationship is *not* so obvious (In fact, it's much easier to make something more complicated than easy, since people are more impressed by complexity than simplicity.)

Getting Spaced Out

Since spaces are used to separate multiple fields, it follows that spaces cannot simply be included in facts. For example,

```
CLIPS> (clear)
CLIPS> (assert (animal-is walrus))
<Fact-1>
CLIPS> (assert ( animal-is walrus ))
<Fact-1>
CLIPS> (assert (  animal-is  walrus  ))
<Fact-1>
CLIPS> (facts)
f-1     (animal-is walrus)
For a total of 1 fact.
CLIPS>
```

Only one fact, (animal-is walrus), is asserted since CLIPS ignores white space and considers all these facts equivalent. Thus, CLIPS responds with <Fact-1> when you try to enter the last two duplicate facts. CLIPS normally does not allow duplicate facts to be entered unless you change the set-fact-duplication setting.

If you want to include spaces in a fact, you must use double quotes. For example,

```
CLIPS> (clear)
CLIPS> (assert (animal-is "duck"))
<Fact-1>
CLIPS> (assert (animal-is "duck "))
<Fact-2>
CLIPS> (assert (animal-is " duck"))
<Fact-3>
CLIPS> (assert (animal-is " duck "))
<Fact-4>
CLIPS> (facts)
```

```
f-1      (animal-is "duck")
f-2      (animal-is "duck ")
f-3      (animal-is " duck")
f-4      (animal-is " duck ")
For a total of 4 facts.
CLIPS>
```

Note that the spaces make each of these facts different to CLIPS although the meaning is the same to a person.

What if you want to include the double quotes in a field? The correct way to put double quotes in a fact is with the **backslash**, "\", as the following example shows.

```
CLIPS> (clear)
CLIPS> (assert (single-quote "duck"))
<Fact-1>
CLIPS> (assert (double-quote "\"duck\""))
<Fact-2>
CLIPS> (facts)
f-1      (single-quote "duck")
f-2      (double-quote ""duck"")
For a total of 2 facts.
CLIPS>
```

Retract that Fact

Now that you know how to put facts into the fact-list, it's time to learn how to remove them. Removing facts from the fact-list is called *retraction* and is done with the retract command. To retract a fact, you must specify the fact index of the fact as the argument of retract. For example, set up your fact-list as follows.

```
CLIPS> (clear)
CLIPS> (assert (animal-is duck))
<Fact-1>
CLIPS> (assert (animal-sound quack))
<Fact-2>
CLIPS> (assert (The duck says "Quack."))
<Fact-3>
CLIPS> (facts)
f-1      (animal-is duck)
```

```
f-2      (animal-sound quack)
f-3      (The duck says "Quack.")
For a total of 3 facts.
CLIPS>
```

To remove the last fact with index f-3, enter the retract command and then check your facts as follows.

```
CLIPS> (retract 3)
CLIPS> (facts)
f-1      (animal-is duck)
f-2      (animal-sound quack)
For a total of 2 facts.
CLIPS>
```

What happens if you try to retract a fact that's already retracted, or a non-existent fact? Let's try it and see.

```
CLIPS> (retract 3)
[PRNTUTIL1] Unable to find fact f-3.
CLIPS>
```

Notice that CLIPS issues an error message if you try to retract a non-existent fact. The moral of this is that you can't take back what you haven't given.

Now let's retract the other facts as follows.

```
CLIPS> (retract 2)
CLIPS> (facts)
f-1      (animal-is duck)
For a total of 1 fact.
CLIPS> (retract 1)
CLIPS> (facts)
CLIPS>
```

You can also retract multiple facts at once, as shown by the following.

```
CLIPS> (clear)
CLIPS> (assert (animal-is duck))
<Fact-1>
CLIPS> (assert (animal-sound quack))
<Fact-2>
```

```
CLIPS> (assert (The duck says "Quack."))
<Fact-3>
CLIPS> (retract 1 3)
CLIPS> (facts)
f-2     (animal-sound quack)
For a total of 1 fact.
CLIPS>
```

To retract multiple facts, just list the fact-id numbers in the (retract) command.

You can just use (**retract ***) to retract all the facts, where the "*****" indicates *all* .

```
CLIPS> (clear)
CLIPS> (assert (animal-is duck))
<Fact-1>
CLIPS> (assert (animal-sound quack))
<Fact-2>
CLIPS> (assert (The duck says "Quack."))
<Fact-3>
CLIPS> (facts)
f-1     (animal-is duck)
f-2     (animal-sound quack)
f-3     (The duck says "Quack.")
For a total of 3 facts.
CLIPS> (retract *)
CLIPS> (facts)
CLIPS>
```

Watch that Fact

CLIPS provides several commands to help you debug programs. One command allows you to continuously **watch facts** being asserted and retracted. This is more convenient than having to type in a (facts) command over and over again and trying to figure out what's changed in the fact-list.

To start watching facts, enter the command (watch facts) as shown in the following example.

```
CLIPS> (clear)
CLIPS> (watch facts)
CLIPS> (assert (animal-is duck))
```

```
==> f-1      (animal-is duck)
<Fact-1>
CLIPS>
```

The **right double arrow** symbol, **==>**, means that a fact is *entering* memory while the **left double arrow** indicates a fact is *leaving* memory, as shown following.

```
CLIPS> (reset)
<== f-1      (animal-is duck)
CLIPS> (assert (animal-is duck))
==> f-1      (animal-is duck)
<Fact-1>
CLIPS> (retract 1)
<== f-1      (animal-is duck)
CLIPS> (facts)
CLIPS>
```

The (watch facts) command provides a record that shows the **dynamic**; or changing state of the fact-list. In contrast, the (facts) command show the **static** state of the fact-list since it displays the current contents of the fact-list. To turn off watching facts, enter (**unwatch facts**).

There are a number of things you can watch. These include the following, which are described in more detail in the *CLIPS Reference Manual*. The **comment** in CLIPS begins with a **semicolon**. Everything after the semicolon is ignored by CLIPS.

```
(watch facts)
; instances used with objects
(watch instances)
; slots used with objects
(watch slots)
(watch rules)
(watch activations)
; messages used with objects
(watch messages)
; message-handlers used with objects
(watch message-handlers)
(watch generic-functions)
(watch methods)
(watch deffunctions)
; compilations are watched by default
```

```
(watch compilations)
(watch statistics)
(watch globals)
(watch focus)
; all watches everything
(watch all)
```

As you use more of the capabilities of CLIPS, you'll find these (watch) commands very helpful in debugging. To turn off a (watch) command, enter an **unwatch** command. For example, to turn off watching compilations, enter (unwatch compilations).

Chapter 2

Following the Rules

If you want to get anywhere in life, don't break the rules — make the rules!

Making Good Rules

To accomplish useful work, an expert system must have rules as well as facts. Since you've seen how facts are asserted and retracted, it's time to see how rules work. A rule is similar to an IF THEN statement in a procedural language like Java, C, or Ada. An IF THEN rule can be expressed in a mixture of natural language and computer language as follows:

```
IF certain conditions are true
THEN execute the following actions
```

Another term for the above statement is **pseudocode**, which literally means *false code*. While pseudocode cannot be directly executed by the computer, it serves as a very useful guide to writing executable code. Pseudocode is also helpful in documenting rules. A translation of rules from natural language to CLIPS is not very difficult if you keep this IF THEN analogy in mind. As your experience with CLIPS grows, you'll find that writing rules in CLIPS becomes easy. You can either type rules directly into CLIPS or load rules in from a file of rules created by a text editor.

The pseudocode for a rule about duck sounds might be

```
IF the animal is a duck
THEN the sound made is quack
```

The following is a fact, and a rule named *duck* which is the pseudocode above expressed in CLIPS syntax. The name of the rule follows immediately after the keyword *defrule*. Although you can enter a rule on a single line, it's customary to put different parts on separate lines to aid readability and editing.

```
CLIPS> (unwatch facts)
CLIPS> (clear)
CLIPS> (assert  (animal-is duck))
<Fact-1>
```

```
CLIPS>
(defrule duck
    (animal-is duck)
=>
    (assert (sound-is quack)))
CLIPS>
```

If you type in the rule correctly as shown, you should see the CLIPS prompt reappear. Otherwise, you'll see an error message. If you get an error message, it is likely that you misspelled a keyword or left out a parenthesis. Remember, the number of left and right parentheses always must match in a statement.

The same rule is shown following with comments added to match the parts of the rule. Also shown is the optional **rule-header** comment in quotes, "Here comes the quack". There can be only one rule-header comment and it must be placed after the rule name and before the first **pattern**. Although we're only discussing pattern matching against facts now, more generally a pattern can be matched against a **pattern entity**. A *pattern entity* is either a fact or an instance of a user-defined class. Pattern matching on objects will be discussed later.

CLIPS tries to match the pattern of the rule against a pattern entity. Of course, white space consisting of spaces, tabs, and carriage returns may be used to separate the elements of a rule to improve readability. Other comments begin with a semicolon and continue until the carriage return key is pressed to terminate a line. Comments are ignored by CLIPS.

```
; Rule header
(defrule duck
    ; Comment
    "Here comes the quack"
    ; Pattern
    (animal-is duck)
=> ; THEN arrow
    ; Action
    (assert (sound-is quack)))
```

📌   *Only one rule name can exist at one time in CLIPS.*

Entering the same rule name, in this case "duck", will replace any existing rule with that name. That is, while there can be many rules in CLIPS, there can be only one rule which is named "duck". This is analogous to other programming languages in which only one procedure name can be used to uniquely identify a procedure.

The general syntax of a rule is shown following.

```
(defrule rule_name "optional_comment"
    (pattern_1)    ; Left-Hand Side (LHS)
    (pattern_2)    ; of the rule consisting
        .          ; of elements before
        .          ; the "=>"
        .
    (pattern_N)
=>
    (action_1)     ; Right-Hand Side (RHS)
    (action_2)     ; of the rule consisting
        .          ; of elements after
        .          ; the "=>". The last ")"
        .          ; balances the opening
    (action_M))    ; "(" to the left of
                   ; "defrule". Be sure all
                   ; your parentheses
                   ; balance or you will
                   ; get error messages
```

The entire rule must be surrounded by parentheses. Each of the rule patterns and **actions** must be surrounded by parentheses. An action is actually a function which typically has no **return value**, but performs some useful action, such as an (assert) or (retract). For example, an action might be (assert (duck)). Here the function name is "assert" and its argument is "duck". Notice that we don't want any return value such as a number. Instead, we want the fact (duck) to be asserted. 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 (printout).

A rule often has multiple patterns and actions. The number of patterns and actions do not have to be equal, which is why different indices, N and M, were chosen for the rule patterns and actions.

Zero or more patterns may be written after the rule header. Each pattern consists of one or more fields. In the duck rule, the pattern is (animal-is duck), where the fields are "animal-is" and "duck". CLIPS attempts to match the patterns of rules against facts in the fact-list. If all the patterns of a rule match facts, the rule is **activated** and put on the **agenda**. The agenda is a collection of **activations** which are those rules which match pattern entities. Zero or more activations may be on the agenda.

The symbol "**=>**" that follows the patterns in a rule is called an **arrow**. The arrow represents the beginning of the THEN part of an IF-THEN rule (and may be read as "implies").

The last part of a rule is the list of zero or more actions that will be executed when the rule **fires**. In our example, the one action is to assert the fact (sound-is quack). The term *fires* means that CLIPS has selected a certain rule for execution from the agenda.

A program will cease execution when no activations are on the agenda.

When multiple activations are on the agenda, CLIPS automatically determines which activation is appropriate to fire. CLIPS orders the activations on the agenda in terms of increasing priority or **salience**.

The part of the rule before the arrow is called the left-hand side (**LHS**) and the part of the rule after the arrow is called the right-hand side (**RHS**). If no patterns are specified, CLIPS automatically activates the rule when a **(reset)** command is entered.

Let's Get Quacking

CLIPS always executes the actions on the RHS of the highest priority rule on the agenda. This rule is then removed from the agenda and the actions of the new highest salience rule is executed. This process continues until there are no more activations or a command to stop is encountered.

You can check what's on the agenda with the **agenda** command. For example,

```
CLIPS> (agenda)
0       duck: f-1
For a total of 1 activation.
CLIPS>
```

The first number "0" is the salience of the "duck" activation, and "f-1" is the fact-identifier of the fact, (animal-is duck), which matches the activation. If the salience of a rule is not declared explicitly, CLIPS assigns it the default value of zero, where the possible salience values range from -10,000 to 10,000. In this book, we'll use the definition of the term **default** as meaning the *standard way*.

If there is only one rule on the agenda, that rule will fire. Since the LHS pattern of the duck-sound rule is

```
(animal-is duck)
```

this pattern will be satisfied by the fact (animal-is duck) and so the duck-sound rule should fire.

Each field of the pattern is said to be a **literal constraint**. The term **literal** means having a constant value, as opposed to a *variable* whose value is expected to change. In this case, the literals are "animal-is" and "duck".

To make a program run, just enter the **run** command. Type (run) and press the carriage return key. Then do a (facts) to check that the fact was asserted by the rule.

```
CLIPS> (run)
CLIPS> (facts)
f-1     (animal-is duck)
f-2     (sound-is quack)
For a total of 2 facts.
CLIPS>
```

Before going on, let's save the duck rule with the **save** command so that you don't have to type it in again (if you haven't already saved it in an editor). Just enter a command such as

```
(save "duck.clp")
```

to save the rule from CLIPS memory to disk and name the file "duck.clp" where the ".clp" is simply a convenient extension to remind us this is a CLIPS source code file. Note that saving the code from CLIPS memory like this will only preserve the optional rule-header comment in quotes and not any semicolon comments.

Kick your Duck

An interesting question may occur to you at this time. What if you (run) again? There is a rule and a fact which satisfies the rule, so the rule should fire. However, if you try this and (run) again, you'll see that the rule won't fire. This may be somewhat frustrating. However, before you do something drastic to ease your frustration — like kicking your pet duck — you need to know a little more about some basic principles of expert systems.

A rule is activated if its patterns are matched by a

1. a brand new pattern entity that did not exist before or,

2. a pattern entity that did exist before but was retracted and reasserted, i.e., a "clone" of the old pattern entity, and thus now a new pattern entity.

The rule, and indices of the matching patterns, is the activation. If either the rule or the pattern entity, or both change, the activation is removed. An activation may also be removed by a command or an action of another rule that fired before and removed the conditions necessary for activation.

The Inference Engine sorts the activations according to their salience. This sorting process is called **conflict resolution** because it eliminates the conflict of deciding which rule should fire next. CLIPS executes the RHS of the rule with the highest salience on the agenda, and removes the activation. This execution is called firing the rule in analogy with the firing of a neuron. A neuron emits a voltage pulse when an appropriate stimulus is applied. After a neuron fires, it undergoes **refraction** and cannot fire again for a certain period of time. Without refraction, neurons would just keep firing over and over again on exactly the same stimulus.

Without refraction, expert systems always would be caught in trivial loops. That is, as soon as a rule fired, it would keep firing on that same fact over and over again. In the real world, the stimulus that caused the firing eventually would disappear. For example, a real duck might swim away or get a job in the movies. However, in the computer world, once data is stored, it stays there until explicitly removed or the power is turned off.

The following example shows activations and firing of a rule. Notice that the (watch) commands are used to more carefully show every fact and activation. The arrow going to the right means an entering fact or activation while an arrow to the left would mean an exiting fact or activation.

```
; Comments in blue/italics have been
; added for explanation. You will
; not see these in the actual output
CLIPS> (clear)
CLIPS>
(defrule duck
   (animal-is duck)
=>
   (assert (sound-is quack)))
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (assert (animal-is duck))
==> f-1     (animal-is duck)
; Activation salience is 0 by default,
; then rule name:pattern entity index
==> Activation 0      duck: f-1
```

```
<Fact-1>
; Notice that duplicate fact
; cannot be entered
CLIPS> (assert (animal-is duck))
<Fact-1>
CLIPS> (agenda)
0       duck: f-1
For a total of 1 activation.
CLIPS> (run)
==> f-2      (sound-is quack)
; Nothing on agenda after rule fires
; Even though fact matches rule,
; refraction will not allow this
; activation because the rule already
; fired on this fact
CLIPS> (agenda)
CLIPS> (facts)
f-1     (animal-is duck)
f-2     (sound-is quack)
For a total of 2 facts.
CLIPS> (run)
CLIPS>
```

You can make the rule fire again if you retract the fact and then assert it as a new fact.

Show Me the Rules

Sometimes you may want to see a rule while you're in CLIPS. There's a command called **ppdefrule** – the pretty print rule – that prints a rule. To see a rule, specify the rule name as an argument to *ppdefrule*. For example,

```
CLIPS> (ppdefrule duck)
(defrule MAIN::duck
   (animal-is duck)
   =>
   (assert (sound-is quack)))
CLIPS>
```

CLIPS puts different parts of the rule on different lines for the sake of readability. The patterns before the arrow are still considered the LHS and the actions after the arrow are still considered the RHS of the rule. The term *MAIN* refers to the MAIN module that this rule is in by default. You can define modules to put rules in analogous to the statements that may be put in different packages, modules, procedures, or functions of other programming languages. The use of modules make it easier to write expert systems having many rules since these may be grouped together with their own agendas for each module. For more information, see the *CLIPS Reference Manual*.

What if you want to print a rule but can't remember the name of the rule? No problem. Just use the **rules** command in response to a CLIPS prompt and CLIPS will print out the names of all the rules. For example, enter

```
CLIPS> (rules)
duck
For a total of 1 defrule.
CLIPS>
```

Write to Me

Besides asserting facts in the RHS of rules, you also can print out information using the **printout** function. CLIPS also has a carriage return/linefeed keyword called **crlf** which is very useful in improving the appearance of output by formatting it on different lines. For a change, the *crlf* is not included in parentheses. As an example,

```
CLIPS>
(defrule duck
    (animal-is duck)
=>
    ;; Be sure to type in the "t"
    (printout t "quack" crlf))
==> Activation 0      duck: f-1
CLIPS> (run)
quack
CLIPS>
```

The output is the text within the double quotes. Be sure to type the letter "**t**" following the printout command. This tells CLIPS to send the output to the **standard output device** of your computer. Generally, the standard output device is your terminal (hence the letter "t" after

printout). However, this may be redefined so that the standard output device is some other device, such as a modem or disk.

Other Features

The **declare salience** command provides explicit control over which rules will be put on the agenda. You must be careful in using this feature too freely lest your program become too controlled. One way to make a rule fire again is to force the rule to be re-activated by the **refresh** rule command.

The **load** command loads in the rule that you had previously saved to disk in the file "duck.clp" or whatever name and directory that you had saved it under. You can load a file of rules made on a text editor into CLIPS using the load command.

A faster way to load files is to first save them in a machine readable binary format with the save binary command called **bsave**. The load binary command, **bload**, can then be used to read these binary rules into CLIPS memory much faster since the files do not have to be re-interpreted by CLIPS.

Two other useful commands allow you to save and load facts using a file. These are **save-facts** and **load-facts**. The (save-facts) will save all the facts in the *fact-list to a file* while (load-facts) will load in the facts from a *file into the fact-list*.

The **batch** command allows you to execute commands from a file as if they were typed in at the top-level. Another useful command provides an interface to your operating system. The **system** command allows the execution of operating system commands or executables within CLIPS. For more information on all these topics, see the *CLIPS Reference Manual*.

Chapter 3

Adding Details

It's not the big picture that is the problem—it's the details

In the first two chapters, you learned the fundamentals of CLIPS. Now you will see how to build on that foundation to create more powerful programs.

Stop And Go

Until now, you've only seen the simplest type of program consisting of just one rule. However, expert systems consisting of only one rule are not very useful. Practical expert systems may consist of hundreds or thousands of rules. Let's now take a look at an application requiring multiple rules.

Suppose you wanted to write an expert system to determine how a mobile robot should respond to a traffic light. It is best to write this type of problem using multiple rules. For example, the rules for the red and green light situations can be written as follows.

```
(defrule red-light
    (light red)
=>
    (printout t "Stop" crlf))

(defrule green-light
    (light green)
=>
    (printout t "Go" crlf))
```

After the rules have been entered into CLIPS, assert a fact (light red) and run. You'll see "Stop" printed. Now assert a (light green) fact and run. You should see "Go" printed.

Take a Walk

If you think about it, other possibilities beside the simple red, green, and yellow cases exist. Some traffic lights also have a green arrow for protected left turns. Some have a hand that lights up to

indicate whether a person can walk or not. Some have signs that say walk or don't walk. So depending on whether our robot is walking or driving, it may have to pay attention to different signs.

The information about walking or driving must be asserted in addition to information about the status of the light. Rules can be made to cover these conditions, but they must have more than one pattern. For example, suppose we want a rule to fire if the robot is walking and if the walk-sign says walk. A rule could be written as follows:

```
(defrule take-a-walk
    (status walking)
    (walk-sign walk)
=>
    (printout t "Go" crlf))
```

The above rule has two patterns. Both patterns must be satisfied by facts in the fact-list for the rule to fire. To see how this works, enter the rule and then assert the facts (status walking) and (walk-sign walk). When you (run), the program will print out "Go" since both patterns are satisfied and the rule is fired.

You can have any number of patterns or actions in a rule. The important point to realize is that the rule is placed on the agenda only if *all* the patterns are satisfied by facts. This type of restriction is called a **logical AND conditional element** (**CE**) in reference to the AND relation of Boolean logic. An AND relation is said to be true only if all its conditions are true.

Because the patterns are of the logical AND type, the rule will not fire if only one of the patterns is satisfied. All facts must be present before the LHS of a rule is satisfied and the rule is placed on the agenda.

A Question of Strategy

The word **strategy** was originally a military term for the planning and operations of warfare. Today, the term *strategy* is commonly used in business (because business is war) to refer to the high-level plans of an organization in achieving its goals, e.g., "Make a lot of money by selling more greasy hamburgers than anyone else in the world!"

In expert systems, one use of the term *strategy* is in conflict resolution of activations. Now you might say, "Well, I'll just design my expert system so that only one rule can possibly be activated at one time. Then there is no need for conflict resolution." The good news is that if you succeed, conflict resolution is indeed unnecessary. The bad news is that this success proves that your

application can be well represented by a sequential program. So you should have coded it in C, Java, or Ada in the first place and not bothered writing it as an expert system.

CLIPS offers seven different modes of conflict resolution: depth, breadth, LEX, MEA, complexity, simplicity, and random. It's difficult to say that one is clearly better than another without considering the specific application. Even then, it may be difficult to judge which is "best." For more information on the details of these strategies, see the *CLIPS Reference Manual*.

The **depth strategy** is the standard **default strategy** of CLIPS. The default setting is automatically set when CLIPS is first started. Afterwards, you can change the default setting. In the depth strategy, new activations are placed on the agenda after activations with higher salience, but before activations with equal or lower salience. All this simply means is that the agenda is ordered from highest to lowest salience.

In this book, all discussions and examples will assume depth strategy.

Now that all these different optional settings are available, be sure that before you run an expert system developed by someone else, that your settings are the same as theirs. Otherwise, you may find the operation is inefficient or even incorrect. In fact, it's a good idea to explicitly encode all the settings in any system that you develop so that it will be configured properly.

Gimme Deffacts

As you work with CLIPS, you may become tired of typing in the same assertions from the top-level. If you are going to use the same assertions every time a program is run, you can first load assertions from a disk using a batch file. An alternative way to enter facts is by using the define facts keyword, **deffacts**. For example,

```
CLIPS> (unwatch facts)
CLIPS> (unwatch activations)
CLIPS> (clear)
CLIPS>
(deffacts walk "Some facts about walking"
   ; status fact to be asserted
   (status walking)
   ; walk-sign fact to be asserted
   (walk-sign walk))
CLIPS> (reset)
; reset causes facts from
; deffacts to be asserted
```

```
CLIPS> (facts)
f-1     (status walking)
f-2     (walk-sign walk)
For a total of 2 facts.
CLIPS>
```

The required name of this deffacts statement, *walk*, follows the deffacts keyword. Following the name is an optional comment in double quotes. Like the optional comment of a rule, the (deffacts) comment will be retained with the (deffacts) after it's been loaded by CLIPS. After the name or comment are the facts that will be asserted in the fact-list. The facts in a deffacts statement are asserted using the CLIPS (reset) command.

The (reset) has an advantage compared to a (clear) command in that (reset) doesn't get rid of all the rules. The (reset) leaves your rules intact. Like (clear), it removes all activated rules from the agenda and also removes all old facts from the fact-list. Giving a (reset) command is a recommended way to start off program execution, especially if the program has been run before and the fact-list is cluttered with old facts.

In summary, the (reset) does two things for facts.

1. It removes existing facts from the fact-list, which may remove activated rules from the agenda.

2. It asserts facts from existing (deffacts) statements.

Actually, the (reset) also does corresponding operations on objects. It deletes instances and asserts instances from **definstances**.

Selective Elimination

The **undeffacts** command excises a (deffacts) from asserting facts by eliminating the deffacts from memory. For example,

```
CLIPS> (undeffacts walk)
CLIPS> (reset)
CLIPS> (facts)
CLIPS>
```

This example demonstrates how the (deffacts) walk has been excised. To restore a deffacts statement after an (undeffacts) command, you must enter the deffacts statement again. In addition to facts, CLIPS also allows you to eliminate rules selectively by using the **undefrule**.

Watch It!

You can **watch rules** firing and **watch activations** on the agenda. The **watch statistics** prints information about the number of rules fired, run time, rules per second, mean number of facts, maximum number of facts, mean number of activations, and maximum number of activations. The statistics information may be useful in **tuning up** an expert system to optimize its speed. Another command, called **watch compilations**, shows information when rules are being loaded. The **watch all** command will watch everything.

Printing of watch information to the screen or to disk with the **dribble** command will slow down your program somewhat because CLIPS uses more time to print or to save to disk. The **dribble-on** command will store all input and output entered at the command prompt to a disk file until the **dribble-off** command is entered. This is convenient in providing a permanent record of everything that happens. These commands are as follows.

```
(dribble-on <filename>)
(dribble-off <filename>)
```

Another useful debugging command is (run) which takes an optional argument of the number of rule firings. For example, a (run 21) command would tell CLIPS to run the program and then stop after 21 rule firings. A (run 1) command allows you to step through a program firing one rule at a time.

Just like many other programming languages, CLIPS also gives you the capability of setting **breakpoints**. A breakpoint is simply an indicator to CLIPS to stop execution just prior to executing a specified rule. A breakpoint is set by the **set-break** command. The **remove-break** command will remove a breakpoint that has been set. The **show-breaks** will list all the rules which have breakpoints set. The syntax of these commands for the argument <rulename> is shown following.

```
(set-break <rulename>)
(remove-break <rulename>)
(show-breaks)
```

A Good Match

You may encounter a situation in which you are certain a rule should be activated but isn't. While it is possible that this is due to a bug in CLIPS, it's not very likely because of the great skill of the people who programmed CLIPS (NOTE: PAID COMMERCIAL ANNOUNCEMENT FOR THE DEVELOPERS).

In most cases, the problem occurs because of the way that you wrote the rule. As an aid to debugging, CLIPS has a command called **matches** that can tell you which patterns in a rule match facts. Patterns which do not match prevent the rule from becoming activated. One common reason that a pattern won't match a fact results from misspelling an element in the pattern or in the assertion of the fact.

The argument of (matches) is the name of the rule to be checked for matches. To see how (matches) works, first (clear), then enter the following rule.

```
(defrule take-a-vacation
    ; Conditional element 1
    (work done)
    ; Conditional element 2
    (money plenty)
    ; Conditional element 3
    (reservations made)
=>
    (printout t "Let's go!!!" crlf))
```

The following shows how (matches) is used. Enter the commands as shown. Notice that (watch facts) is turned on. This is a good idea when you are asserting facts manually since it gives you an opportunity to check the spelling of facts.

```
CLIPS> (watch facts)
CLIPS> (assert (work done))
==> f-1     (work done)
<Fact-1>
CLIPS> (matches take-a-vacation)
Matches for Pattern 1
f-1
Matches for Pattern 2
 None
Matches for Pattern 3
 None
; CE is conditional element
Partial matches for CEs 1 - 2
 None
Partial matches for CEs 1 - 3
 None
Activations
```

```
  None
; The return value indicates the total patterns
; matched, the total partial matches, and the
; total activations
(1 0 0)
CLIPS>
```

The fact with fact-identifier f-1 matches the first pattern or conditional element in the rule and is reported by (matches). Given that a rule has N patterns, the term **partial matches** refers to any set of matches of the first N-1 patterns with facts. That is, the partial matches begin with the first pattern in a rule and end with any pattern up to but not including the last (Nth) pattern. As soon as one partial match cannot be made, CLIPS does not check any further. For example, a rule with four patterns would have partial matches of the first and second patterns and also of the first, second, and third patterns. If all N patterns match, the rule will be activated.

Other Features

Some additional commands are useful with deffacts. For example, the command **list-deffacts** will list the names of currently loaded deffacts in CLIPS. Another useful command is **ppdeffacts** which prints the facts stored in a deffacts.

Other functions allow you to manipulate strings easily:

assert-string: Performs a string assertion by taking a string as argument and asserted as a non-string fact.

str-cat: Constructs a single-quoted string from individual items by string concatenation.

str-index: Returns a string index of the first occurrence of a substring.

sub-string: Returns a substring from a string.

str-compare: Performs a string compare.

str-length: Returns the string length which is the length of a string.

sym-cat: Returns a concatenated symbol.

If you want to printout a multifield variable without parentheses, the simplest way is by using the string implode function, implode$.

Variable Interests

Nothing changes more than change

The type of rules that you've seen so far illustrates simple matching of patterns to facts. In this chapter, you'll learn very powerful ways to match and manipulate facts.

Let's Get Variable

Just as with other programming languages, CLIPS has **variables** to store values. Unlike a fact, which is **static** or unchanging, the contents of a variable are **dynamic** as the values assigned to it change. In contrast, once a fact is asserted, it's fields can only be modified by retracting and asserting a new fact with the changed fields, Even the *modify* action (described later in the chapter on deftemplate) acts by retracting and asserting a modified fact, as you can see by checking the fact-index.

The name of a variable, or **variable identifier**, is always written by a question mark followed by a symbol that is the name of the variable. The general format is

```
?<variable-name>
```

Global variables, to be described in more detail later, have a slightly different syntax.

Just as in other programming languages, variable names should be meaningful for good style. Some examples of valid variable names follow.

```
?x
?noun
?color
?sensor
?valve
?ducks-eaten
```

Before a variable can be used, it should be assigned a value. As an example of a case where a value is *not* assigned, try to enter the following and CLIPS will respond with the error message shown.

```
CLIPS> (unwatch all)
CLIPS> (clear)
CLIPS>
(defrule test
=>
   (printout t ?x crlf))

[PRCCODE3] Undefined variable x referenced in RHS of defrule.

ERROR:
(defrule MAIN::test
   =>
   (printout t ?x crlf))
CLIPS>
```

CLIPS gives an error message when it cannot find a value **bound** to ?x. The term *bound* means the assignment of a value to a variable. Only global variables are bound in *all* rules. All other variables are only bound *within* a rule. Before and after a rule fires, non-global variables are not bound and so CLIPS will give an error message if you try to query a non-bound variable.

Be Assertive

One common use of variables is to match a value on the LHS and then assert this bound variable on the RHS. For example, enter

```
(defrule make-quack
   (duck-sound ?sound)
=>
   (assert (sound-is ?sound)))
```

Now assert (duck-sound quack), then (run) the program. Check the facts and you'll see that the rule has produced (sound-is quack) because the variable ?sound was bound to quack.

Of course, you also can use a variable more than once. For example, enter the following. Be sure to do a (reset) and assert (duck-sound quack) again.

```
(defrule make-quack
   (duck-sound ?sound)
=>
   (assert (sound-is ?sound ?sound)))
```

When the rule fires, it will produce (sound-is quack quack) since the variable ?sound is used twice.

What the Duck Said

Variables also are used commonly in printing output, as in

```
(defrule make-quack
    (duck-sound ?sound)
=>
    (printout t "The duck said "
               ?sound crlf))
```

Do a (reset), enter this rule, and assert the fact and then (run) to find out what the duck said. How would you modify the rule to put double quotes around quack in the output?

More than one variable may be used in a pattern, as the following example shows.

```
CLIPS> (clear)
CLIPS>
(defrule whodunit
    (duckshoot ?hunter ?who)
=>
    (printout t ?hunter " shot "
               ?who crlf))
CLIPS> (assert (duckshoot Brian duck))
<Fact-1>
; Duck dinner tonight!
CLIPS> (run)
Brian shot duck
CLIPS> (assert (duckshoot duck Brian))
<Fact-2>
; Brian dinner tonight!
CLIPS> (run)
duck shot Brian
; Missing third field
CLIPS> (assert (duckshoot duck))
<Fact-3>
; Rule doesn't fire,
; no output
```

```
CLIPS> (run)
CLIPS>
```

Notice what a big difference the order of fields makes in determining who shot who. You can also see that the rule did *not* fire when the fact (duckshoot duck) was asserted. The rule was not activated because no field of the fact matched the second pattern constraint, ?who.

The Happy Bachelor

Retraction is very useful in expert systems and usually done on the RHS rather than at the top-level. Before a fact can be retracted, it must be specified to CLIPS. To retract a fact from a rule, the **fact-address** first must be bound to a variable on the LHS.

There is a big difference between binding a variable to the contents of a fact and binding a variable to the fact-address. In the examples that you've seen such as (duck-sound ?sound), a variable was bound to the value of a field. That is, ?sound was bound to quack. However, if you want to remove the fact whose contents are (duck-sound quack), you must first tell CLIPS the *address* of the fact to be retracted.

The fact-address is specified using the **left arrow**, "<-". To create this, just type a "<" symbol followed by a "-". As an example of fact retraction from a rule,

```
CLIPS> (clear)
CLIPS> (assert (bachelor Dopey))
<Fact-1>
CLIPS> (facts)
f-1     (bachelor Dopey)
For a total of 1 fact.
CLIPS>
(defrule get-married
   ?duck <- (bachelor Dopey)
=>
   (printout t "Dopey is now happily married "
               ?duck crlf)
   (retract  ?duck))
CLIPS> (run)
Dopey is now happily married <Fact-1>
CLIPS> (facts)
CLIPS>
```

Notice that the (printout) prints the fact-index of ?duck, <Fact-1>, since the left arrow bound the address of the fact to ?duck. Also, there is no fact (bachelor Dopey) because it has been retracted.

Variables can be used to pick up a fact value at the same time as an address, as shown in the following example. For convenience, a (deffacts) has also been defined.

```
CLIPS> (clear)
CLIPS>
(defrule marriage
    ?duck <- (bachelor ?name)
=>
    (printout t ?name
                " is now happily married"
                crlf)
    (retract  ?duck))
CLIPS>
(deffacts good-prospects
    (bachelor Dopey)
    (bachelor Dorky)
    (bachelor Dicky))
CLIPS> (reset)
CLIPS> (run)
Dicky is now happily married
Dorky is now happily married
Dopey is now happily married
CLIPS>
```

Notice how the rule fired on *all* facts that matched the pattern (bachelor ?name). CLIPS also has a function called **fact-index** which can be used to return the fact index of a fact address.

It's Not Important

Instead of binding a field value to a variable, the presence of a nonempty field can be detected alone using a **wildcard**. For example, suppose you're running a dating service for ducks, and a duckette asserts that she only dates ducks whose first name is Dopey. Actually, two criteria are in this specification since there is an implication that the duck must have more than one name. So a plain (bachelor Dopey) isn't adequate because there is only one name in the fact.

This type of situation, in which only part of the fact is specified, is very common and very important. To solve this problem, a wildcard can be used to match the Dopeys.

The simplest form of wildcard is called a **single-field wildcard** and is shown by a question mark, "**?**". The "?" is also called a **single-field constraint**. A single-field wildcard stands for *exactly* one field, as shown following.

```
CLIPS> (clear)
CLIPS>
(defrule dating-ducks
    (bachelor Dopey ?)
=>
    (printout t "Date Dopey"
                crlf))
CLIPS>
(deffacts duck
    (bachelor Dicky)
    (bachelor Dopey)
    (bachelor Dopey Mallard)
    (bachelor Dinky Dopey)
    (bachelor Dopey Dinky Mallard))
CLIPS> (reset)
CLIPS> (run)
Date Dopey
CLIPS>
```

The pattern includes a wildcard to indicate that Dopey's last name is not important. So long as the first name is Dopey and there is any last name (but no middle names), the rule will be satisfied and fire. Because the pattern has three fields of which one is a single-field wildcard, only facts of *exactly* three fields can satisfy it. In other words, only Dopeys with exactly two names can satisfy this duckette.

Suppose you want to specify Dopeys with exactly three names? All that you'd have to do is write a pattern like

```
(bachelor Dopey ? ?)
```

or, if only persons with three names whose middle name was Dopey,

```
(bachelor ? Dopey ?)
```

or, if only the last name was Dopey, as in the following:

```
(bachelor ? ? Dopey)
```

Another interesting possibility occurs if Dopey *must* be the first name, but only those Dopeys with two *or* three names are acceptable. One way of solving this problem is to write two rules. For example

```
(defrule eligible
   (bachelor Dopey ?)
=>
   (printout t "Date Dopey" crlf))

(defrule eligible-three-names
   (bachelor Dopey ? ?)
=>
   (printout t "Date Dopey" crlf))
```

Enter and run this and you'll see that Dopeys with both two and three names are printed. Of course, if you don't want anonymous dates, you need to bind the Dopey names with a variable and print them out.

Going Wild

Rather than writing separate rules to handle each field, it's much easier to use the **multifield wildcard**. This is a dollar sign followed by a question mark, "**$?**", and represents *zero or more fields*. Notice how this contrasts with the single-field wildcard which must match exactly one field.

The two rules for dates can now be written in a single rule as follows.

```
CLIPS> (clear)
CLIPS>
(defrule dating-ducks
   (bachelor Dopey $?)
=>
   (printout t "Date Dopey" crlf))
CLIPS>
(deffacts duck
   (bachelor Dicky)
   (bachelor Dopey)
   (bachelor Dopey Mallard)
   (bachelor Dinky Dopey)
   (bachelor Dopey Dinky Mallard))
CLIPS> (reset)
```

```
CLIPS> (run)
Date Dopey
Date Dopey
Date Dopey
CLIPS>
```

Wildcards have another important use because they can be attached to a symbolic field to create a variable such as ?x, $?x, ?name, or $?name. The variable can be a **single-field variable** or a **multifield variable** depending on whether a "?" or "$?" is used on the LHS. Note that on the RHS only a ?x is used, where the "x" can be any variable name. You can think of the "$" as a function whose argument is a single-field wildcard or a single-field variable and returns a multifield wildcard or a multifield variable, respectively.

As an example of a multifield variable, the following version of the rule also prints out the name field(s) of the matching fact because a variable is equated to the name field(s) that match:

```
CLIPS>
(defrule dating-ducks
    (bachelor Dopey $?name)
=>
    (printout t "Date Dopey " ?name crlf))
CLIPS> (reset)
CLIPS> (run)
Date Dopey (Dinky Mallard)
Date Dopey (Mallard)
Date Dopey ()
CLIPS>
```

As you can see, on the LHS, the multifield pattern is *$?name* but is *?name* when used as a variable on the RHS. When you enter and run, you'll see the names of all eligible Dopeys. The multifield wildcard takes care of any number of fields. Also, notice that multifield values are returned enclosed in parentheses.

Suppose you wanted a match of all ducks who had a Dopey somewhere in their name, not necessarily as their first name. The following version of the rule would match all facts with a Dopey in them and then print out the names:

```
CLIPS>
(defrule dating-ducks
    (bachelor $?first Dopey $?last)
=>
```

```
        (printout t "Date "
                    ?first
                    " Dopey "
                    ?last crlf))
  CLIPS> (reset)
  CLIPS> (run)
  Date () Dopey (Dinky Mallard)
  Date (Dinky) Dopey ()
  Date () Dopey (Mallard)
  Date () Dopey ()
  CLIPS>
```

The pattern matches *any* names that have a Dopey anywhere in them.

Single- and multifield wildcards can be combined. For example, the pattern

```
(bachelor ? $? Dopey ?)
```

means that the first and last names can be anything and that the name just prior to the last must be Dopey. This pattern also requires that the matching fact will have *at least four* fields, since the "$?" matches *zero* or more fields and all the others must match *exactly* four.

Although multifield variables can be essential for pattern matching in many cases, their overuse can cause much inefficiency because of increased memory requirements and slower execution.

   As a general rule of style, you should use $? only when you don't know the length of fields. Do not use $? simply as a typing convenience.

The Ideal Bachelor

Variables used in patterns have an important and useful property, which can be stated as follows.

   The first time a variable is bound it retains that value only within the rule, both on the LHS and also on the RHS, unless changed on the RHS.

For example, in the rule below

```
(defrule bound
    (number-1 ?num)
    (number-2 ?num)
=>)
```

If there are some facts

```
f-1      (number-1 0)
f-2      (number-2 0)
f-3      (number-1 1)
f-4      (number-2 1)
```

then the rule can only be activated by the pair f-1, f-2, and the other pair f-3, f-4. That is, fact f-1 cannot match with f-4 because when ?num is bound to 0 in the first pattern, the value of ?num in the second pattern also *must* be 0. Likewise, when ?num is bound to 1 in the first pattern, the value of ?num in the second pattern *must* be 1. Notice that the rule will be activated *twice* by these four facts: one activation for the pair f-1, f-2, and the other activation for the pair f-3, f-4.

As a more practical example, enter the following rule. Notice that the same variable, ?name, is used in both patterns. Before doing a (reset) and (run), also enter a (watch all) command so that you can see what happens during execution.

```
CLIPS> (clear)
CLIPS>
(defrule ideal-duck-bachelor
    (bill big ?name)
    (feet wide ?name)
=>
    (printout t "The ideal duck is "
              ?name crlf))
CLIPS>
(deffacts duck-assets
    (bill big Dopey)
    (bill big Dorky)
    (bill little Dicky)
    (feet wide Dopey)
    (feet narrow Dorky)
    (feet narrow Dicky))
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (reset)
==> f-1      (bill big Dopey)
==> f-2      (bill big Dorky)
==> f-3      (bill little Dicky)
==> f-4      (feet wide Dopey)
```

```
==> Activation 0      ideal-duck-bachelor: f-1,f-4
==> f-5     (feet narrow Dorky)
==> f-6     (feet narrow Dicky)
CLIPS> (run)
The ideal duck is Dopey
CLIPS>
```

When the program is run, the first pattern matches Dopey and Dorky since they both have big bills. The variable ?name is bound to each name. When CLIPS tries to match the second pattern of the rule, only the variable ?name which is bound to Dopey also satisfies the second pattern of (feet wide).

The Lucky Duck

Many situations occur in life where it's wise to do things in a systematic manner. That way, if your expectations don't work out you can try again systematically (such as the common algorithm for finding the Perfect Spouse by getting married over and over again).

One way of being organized is to keep a list. (Note: if you *really* want to impress people, show them a list of your lists.) In our case, we'll keep a list of duck bachelors, with the most likely prospect for matrimony at the front. Once an ideal duck bachelor has been identified, we'll shoot him up to the front of the list as the lucky duck.

The following program shows how this can be done by adding a couple of rules to the ideal-duck-bachelor rule.

```
(defrule ideal-duck-bachelor
   (bill big ?name)
   (feet wide ?name)
=>
   (printout t "The ideal duck is "
              ?name crlf)
   (assert (move-to-front ?name)))

(defrule move-to-front
   ?move-to-front <- (move-to-front ?who)
   ?old-list <-
      (list $?front ?who $?rear)
=>
   (retract ?move-to-front ?old-list)
```

```
      (assert (list ?who ?front ?rear))
      (assert (change-list yes)))

   (defrule print-list
      ?change-list <- (change-list yes)
      (list $?list)
   =>
      (retract ?change-list)
      (printout t "List is : " ?list crlf))

   (deffacts duck-bachelor-list
      (list Dorky Dinky Dicky))

   (deffacts duck-assets
      (bill big Dicky)
      (bill big Dorky)
      (bill little Dinky)
      (feet wide Dicky)
      (feet narrow Dorky)
      (feet narrow Dinky))
```

The original list is given in the duck-bachelor-list deffacts. When the program is run, it will provide a new list of likely candidates.

```
   CLIPS> (unwatch all)
   CLIPS> (reset)
   CLIPS> (run)
   The ideal duck is Dicky
   List is : (Dicky Dorky Dinky)
   CLIPS>
```

Notice the assertion (change-list yes) in the move-to-front rule. Without this assertion, the print-list rule would always fire on the original list. This assertion is an example of a **control fact** made to control the firing of another rule. Control facts are very important in controlling the activation of certain rules, and you should study this example carefully to understand why it's used. Another method of control is modules, as discussed in the *CLIPS Reference Manual*.

The move-to-front rule removes the old list and asserts the new list. If the old list was not retracted, two activations would be on the agenda for the print-list rule but only one would fire. Only one will fire because the print-list rule removes the control fact required for the other

activation of the same rule. You would not know in advance which one would fire, so the old list might be printed instead of the new list.

Chapter 5

Doing It Up In Style

Style today, gone tomorrow

In this chapter, you will learn about a keyword called *deftemplate*, which stands for **define template**. This feature can aid you in writing rules whose patterns have a well-defined structure.

Mr. Wonderful

Deftemplate is analogous to a struct definition in C. That is, the deftemplate defines a group of related fields in a pattern similar to the way in which a C struct is a group of related data. A deftemplate is a list of named fields called **slots**. Deftemplate allows access by name rather than by specifying the order of fields. Deftemplate contributes to good style in expert systems programs and is a valuable tool of software engineering.

A *slot* is a named **single-slot** or **multislot**. A single-slot or simply *slot* contains exactly one field while a multislot contains zero or more fields. Any number of single or multislot slots may be used in a deftemplate. To write a slot, give the field name (attribute) followed by the field value. Note that a multislot slot with one value is strictly not the same as a single-slot slot. As an analogy, think of a cupboard (the multislot) that may contain dishes. A cupboard with one dish is not the same as a dish (single-slot). However, the value of a single-slot slot (or variable) may match a multislot slot (or multislot variable) that has one field.

As an example of a deftemplate relation, consider the attributes of a duck who might be considered a good matrimonial prospect:

name

```
    "Dopey Wonderful"
```

assets

```
    rich
```

age

```
    99
```

A deftemplate may be defined for the relation *prospect* as follows, where white space and comments are used for readability and explanation.

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

In this example, the components of deftemplate are structured as:

- A deftemplate relation name

- Attributes called *fields*

- The field type, which can be any one of the allowed types: symbol, string, number, and others.

- The default for the field value

This particular deftemplate has three single-slot slots called *name*, *assets*, and *age*.

The **deftemplate default values** are inserted by CLIPS when a (reset) is done if no explicit values are defined. For example, enter the deftemplate for prospect after a (clear) command, and assert it as shown.

```
CLIPS> (assert (prospect))
<Fact-1>
CLIPS> (facts)
f-1     (prospect (name "") (assets rich) (age 80))
For a total of 1 fact.
CLIPS>
```

As you can see, CLIPS has inserted the default value of the null string, "", for the name field since that is the default for a STRING. Likewise, the assets and age defaults were also inserted by CLIPS. Different types have different default symbols such as the null string, "", for STRING; the integer 0 for INTEGER; the float 0.0 for FLOAT; and so on. The ?DERIVE keyword selects the appropriate type of constraint for that slot, e.g., the null string , "", for a slot of type STRING.

You can explicitly set the field values, as the following example shows.

```
CLIPS>
(assert (prospect (age 99)
                  (name "Dopey")))
<Fact-2>
CLIPS> (facts)
f-1     (prospect (name "") (assets rich) (age 80))
f-2     (prospect (name "Dopey") (assets rich) (age 99))
For a total of 2 facts.
CLIPS>
```

Note that the order that the fields are typed in does not matter since these are named fields.

In the deftemplate, it's important to realize that NUMBER is *not* a primitive field type like symbol, string, integer, and float. The NUMBER is really a compound type that can be integer *or* float. It is used for cases in which the user doesn't care what type of numbers are stored. An alternative to NUMBER would be specifying the types as follows.

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

Bye-Bye

In general, a deftemplate with N slots has the following general structure:

```
(deftemplate <name>
    (slot-1)
    (slot-2)
       .
       .
       .
    (slot-N))
```

In a deftemplate, the attribute values may be specified more precisely than a simple value such as *80* or *rich*. For example, in this deftemplate, a **type** of value is specified.

The field values can be specified by either explicitly listing them or giving a range of values. The *allowed-values* can be any primitive type such as SYMBOL, STRING, INTEGER, FLOAT and so on. Some examples of deftemplate enumerated values are shown following:

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
```

It doesn't make sense to specify both a numeric range and values allowed for the same deftemplate field. For example, if you specify (allowed-integers 1 4 8), this contradicts a range specification of 1 to 10 by (range 1 10). If the numbers happen to be sequential, such as 1, 2, 3, then you could specify a range which would exactly match: (range 1 3). However, the range would be redundant to the allowed-integers specification. Thus, range and allowed values are

mutually exclusive. That is, if you specify a range, you can't specify the allowed values and *vice versa.* In general, the range attribute cannot be used in conjunction with allowed-values, allowed-numbers, allowed-integers, or allowed-floats.

Without the optional information, the deftemplate and a rule which uses it follows.

```
CLIPS> (clear)
CLIPS>
; name of deftemplate
(deftemplate prospect
   ; name of field
   (slot name
      ; default value of field name
      (default ?DERIVE))
   ; name of field
   (slot assets
      ; default value of field assets
      (default rich))
   ; name of field
   (slot age
      ; default value of field age
      (default 80)))
CLIPS>
(defrule matrimonial_candidate
   (prospect (name ?name)
             (assets ?net_worth)
             (age ?months))
=>
   (printout t "Prospect: "
                ?name crlf
                ?net_worth crlf
                ?months " months old"
                crlf))
CLIPS>
(assert
   (prospect (name "Dopey Wonderful")
             (age 99)))
<Fact-1>
CLIPS> (run)
```

```
Prospect: Dopey Wonderful
rich
99 months old
CLIPS>
```

Notice that the default value of *rich* was used for Dopey since the assets field was not specified in the assert command.

If the assets field is given a specific value such as *poor*, the specified value for *assets* of *poor* overrides the default value of *rich* as shown in the following example about Dopey's penurious nephew.

```
CLIPS> (reset)
CLIPS>
(assert
    (prospect (name "Dopey Notwonderful")
              (assets poor)
              (age 95)))
<Fact-1>
CLIPS> (run)
Prospect: "Dopey Notwonderful"
poor
95 months old
CLIPS>
```

A deftemplate pattern may be used just like any ordinary pattern. For example, the following rule will eliminate undesirable prospects.

```
CLIPS> (undefrule matrimonial_candidate)
CLIPS>
(defrule bye-bye
    ?bad-prospect <-
        (prospect (assets poor)
                  (name ?name))
    =>
    (retract ?bad-prospect)
    (printout t "bye-bye " ?name crlf))
CLIPS> (reset)
CLIPS>
(assert
```

```
       (prospect (name "Dopey Wonderful")
                 (assets rich)))
<Fact-1>
CLIPS>
(assert
       (prospect (name "Dopey Notwonderful")
                 (assets poor)))
<Fact-2>
CLIPS> (run)
bye-bye Dopey Notwonderful
CLIPS>
```

Ain't No Strings on Me

Notice that only single fields were used for the patterns in the examples so far. That is, the field values for *name*, *assets*, and *age*, were all single values. In some types of rules, you may want multiple fields. Deftemplate allows the use of multiple values in a *multislot*.

As an example of multislot, suppose that you wanted to treat the name of the relation *prospect* as multiple fields. This would provide more flexibility in processing prospects since any part of the name could be pattern matched. Shown following is the deftemplate definition using multislot and the revised rule to pattern match on multiple fields. Notice that a multislot pattern, $?name, is now used to match all the fields that make up the name. For convenience, a (deffacts) is also given.

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

```
           (default 80)))          ; better!!!
  CLIPS>
  (defrule happy_relationship
      (prospect (name $?name)
                (assets ?net_worth)
                (age ?months))
  =>
      (printout t "Prospect: "
                   ; Note: not $?name
                   ?name crlf
                   ?net_worth crlf
                   ?months " months old"
                   crlf))
  CLIPS>
  (deffacts duck-bachelor
      (prospect (name Dopey Wonderful)
                (assets rich)
                (age 99)))
  CLIPS> (reset)
  CLIPS> (run)
  Prospect: (Dopey Wonderful)
  rich
  99 months old
  CLIPS>
```

In the output, the parentheses around Dopey's name are put in by CLIPS to indicate that this is a multislot value. If you compare the output from this multislot version to the single-slot version, you'll see that the double quotes around "Dopey Wonderful" are gone. The name *slot* is not a string in the multislot version, so CLIPS treats the name as two independent fields, *Dopey* and *Wonderful*.

What's in a Name

Deftemplate greatly simplifies accessing a specific field in a pattern because the desired field can be identified by its slot name. The **modify** action can be used to modify a fact in one action by specifying one or more template slots to be modified.

As an example, consider the following rules which show what happens when duck-bachelor Dopey Wonderful loses all his fish buying Donald Duck posters and banana fishsplits for his new duckette, Dixie.

```
CLIPS> (undefrule *)
CLIPS>
(defrule make-bad-buys
   ?prospect <- (prospect (name $?name)
                          (assets rich)
                          (age ?months))
=>
   (printout t "Prospect: "
                ; Note: not $?name
                ?name crlf
                "rich" crlf
                ?months " months old"
                crlf crlf)
   (modify ?prospect (assets poor)))
CLIPS>
(defrule poor-prospect
   ?prospect <- (prospect (name $?name)
                          (assets poor)
                          (age ?months))
=>
  (printout t "Ex-prospect: "
                ; Note: not $?name
                ?name crlf
                poor crlf
                ?months " months old"
                crlf crlf))
CLIPS>
(deffacts duck-bachelor
   (prospect (name Dopey Wonderful)
             (assets rich)
             (age 99)))
CLIPS> (reset)
CLIPS> (run)
Prospect: (Dopey Wonderful)
rich
```

```
99 months old

Ex-prospect: (Dopey Wonderful)
poor
99 months old

CLIPS>
```

If you do a (facts) command as follows, you'll see that the f-1 fact originally corresponding to (prospect (assets rich) (age 99) (name Dopey Wonderful)) has been modified and the assets slot has been set to poor.

```
CLIPS> (facts)
f-1      (prospect (name Dopey Wonderful)
                  (assets poor) (age 99))
For a total of 1 fact.
CLIPS>
```

The make-bad-buys rule is activated by a rich prospect as specified by the assets slot. This rule changes the assets to *poor* using the modify action. Notice that the slot *assets* can be accessed by name. Without a deftemplate, it would be necessary to enumerate all the fields by single variables or by using a wildcard, which is less efficient. The purpose of the poor-prospect rule is simply to print out the poor prospects, thus demonstrating that the make-bad-investments rule did indeed modify the assets.

Chapter 6

Being Functional

Functionality is the inverse of style

In this chapter, you will learn more powerful functions for matching patterns and some that are very useful with multifield variables. You also will learn how numeric calculations are done.

Not My Constraint

Let's reconsider the problem of designing an expert system to help a robot cross a street. One rule that you would have follows.

```
(defrule green-light
    (light green)
=>
    (printout t "Walk" crlf))
```

Another rule would cover the case of a red light.

```
(defrule red-light
    (light red)
=>
    (printout t "Don't walk" crlf))
```

A third rule would cover the case in which a walk-sign said not to walk. This would take precedence over a green light.

```
(defrule walk-sign
    (walk-sign-says dont-walk)
=>
    (printout t "Don't walk" crlf))
```

The previous rules are simplified and don't cover all cases such as the breakdown of the traffic-light. For example, what does the robot do if the light is red or yellow and the walk-sign says walk?

A way of handling this case is to use a **field constraint** to restrict the values that a pattern may have on the LHS. The field constraint acts like constraints on patterns.

One type of field constraint is called a **connective constraint**. There are three types of connective constraints. The first is called a **~ constraint**. Its symbol is the **tilde**, "**~**". The ~ constraint acts on the one value that immediately follows it and will *not* allow that value.

As a simple example of the ~ constraint, suppose you wanted to write a rule that would print out "Don't walk" if the light was not green. One approach would be to write rules for every possible light condition, including all possible malfunctions: yellow, red, blinking yellow, blinking red, blinking green, winking yellow, blinking yellow and winking red, and so forth. However, a much easier approach is to use the ~ constraint as shown in the following rule:

```
(defrule walk
    (light ~green)
=>
    (printout t "Don't walk" crlf))
```

By using the ~ constraint, this one rule does the work of many other rules that required specifying each light condition.

Be Cautious

The second connective constraint is the **bar constraint**, "**|**". The "|" connective constraint is used to allow any of a group of values to match.

For example, suppose you wanted a rule that printed out "Be cautious" if the light was yellow or blinking yellow. The following example shows how it's done using the "|" constraint.

```
CLIPS> (clear)
CLIPS>
(defrule cautious
    (light yellow|blinking-yellow)
=>
    (printout t "Be cautious" crlf))
CLIPS> (assert (light yellow))
<Fact-1>
CLIPS> (assert (light blinking-yellow))
<Fact-2>
CLIPS> (agenda)
```

```
0       cautious: f-2
0       cautious: f-1
For a total of 2 activations.
CLIPS>
```

And Away We Go

The third type of connective constraint is the **& connective constraint**. The symbol of the & connective constraint is the **ampersand**, "**&**". The & constraint forces connected constraints to match in union, as you'll see in the following examples. The & constraint normally is used only with the other constraints, otherwise it's not of much practical use. As an example, suppose you want to have a rule that will be triggered by a yellow or blinking-yellow fact. That's easy enough —just use the | connective constraint as you did in a previous example. But suppose that you also want to identify the light color?

The solution is to bind a variable to the color that is matched using the "&" and then print out the variable. This is where the "&" is useful, as shown below.

```
(defrule cautious
    (light ?color&yellow|blinking-yellow)
=>
    (printout t
        "Be cautious because light is "
        ?color crlf))
```

The variable ?color will be bound to whatever color is matched by the field yellow|blinking-yellow.

The "&" also is useful with the "~". For example, suppose you want a rule that triggers when the light is not yellow and not red.

```
(defrule not-yellow-red
    (light ?color&~red&~yellow)
=>
    (printout t "Go, since light is "
                ?color crlf))
```

It's Elementary

Besides dealing with symbolic facts, CLIPS also can perform numeric calculations. However, you should keep in mind that an expert system language like CLIPS is not primarily designed for number-crunching. Although the math functions of CLIPS are very powerful, they are really meant for modification of numbers that are being reasoned about by the application program. Other languages such as FORTRAN are better for number-crunching in which little or no symbolic reasoning is being done. You'll find the computational capability of CLIPS useful in many applications.

CLIPS provides basic arithmetic and math functions +, /, *, -, div, max, min, abs, float, and integer. For more details, see the *CLIPS Reference Manual*.

Numeric expressions are represented in CLIPS according to the style of LISP. In both LISP and CLIPS, a numeric expression that customarily would be written as 2 + 3 must be written in **prefix form**, (+ 2 3). In the prefix form of CLIPS, the function precedes the arguments, and parentheses must surround the numeric expression. The customary way of writing numeric expressions is called **infix form** because the math functions are fixed in between the **arguments**.

Functions can be used on the LHS and the RHS. For example, the following shows how the arithmetic operation of addition is used on the RHS of a rule to assert a fact containing the sum of two numbers ?x and ?y. Note that the comments are in infix notation for your information only since infix cannot be evaluated by CLIPS.

```
CLIPS> (clear)
CLIPS>
(defrule addition
   (numbers ?x ?y)
=>
   ; Add ?x + ?y
   (assert (answer-plus (+ ?x ?y))))
CLIPS> (assert (numbers 2 3))
<Fact-1>
CLIPS> (run)
CLIPS> (facts)
f-1     (numbers 2 3)
f-2     (answer-plus 5)
For a total of 2 facts.
CLIPS>
```

A function can be used on the LHS if an **equal sign**, **=**, is used to tell CLIPS to evaluate the following expression rather than use it literally for pattern matching. The following example shows how the hypotenuse is calculated on the LHS and used to pattern match against some stock items. The **exponentiation**, "******", function is used to square the x and y values. The first argument of exponentiation is the number which is to be raised to the power of the second argument.

```
CLIPS> (clear)
CLIPS>
(deffacts database
    (stock A 2.0)
    (stock B 5.0)
    (stock C 7.0))
CLIPS>
(defrule addition
    (numbers ?x ?y)
                ; Hypotenuse
    (stock ?ID =(sqrt (+ (** ?x 2)
                         (** ?y 2))))
=>
    (printout t "Stock ID=" ?ID crlf))
CLIPS> (reset)
CLIPS> (assert (numbers 3 4))
<Fact-4>
; Stock ID matches
; hypotenuse calculated
CLIPS> (run)
Stock ID=B
CLIPS>
```

Extensive Arguments

Arguments in a numeric expression can be extended beyond two for many of the math functions. The same sequence of arithmetic calculations is performed for more than two arguments. The following example illustrates how three arguments are used. Evaluation proceeds from left to right. Before entering these, however, you may wish to do a (clear) to get rid of any old facts and rules.

```
(defrule addition
    (numbers ?x ?y ?z)
=>
    ; ?x + ?y + ?z
    (assert (answer-plus (+ ?x ?y ?z))))
```

Enter the above program and assert (numbers 2 3 4). After you run, you'll see the following facts. Note that the fact-indices may be different if you've done a (reset) instead of a (clear) before loading this program.

```
CLIPS> (facts)
f-1     (numbers 2 3 4)
f-2     (answer-plus 9)
For a total of 2 facts.
CLIPS>
```

The infix equivalent of a multiple argument CLIPS expression can be expressed as

```
arg [function arg]
```

where the square brackets mean that there can be multiple terms.

Besides the basic math functions, CLIPS has **Extended Math functions** including trig, hyperbolic, and so on. For a complete list, see the *CLIPS Reference Manual.* These are called *Extended Math* functions because they are not considered basic math functions like "+", "-", etc.

Mixed Results

In dealing with expressions, CLIPS tries to keep the mode the same as the arguments. For example,

```
; both integer arguments
; give integer result
CLIPS> (+ 2 2)
4
; both floating-point arguments
; give floating-point result
CLIPS> (+ 2.0 2.0)
4.0
; mixed arguments
; give floating-point result
```

```
CLIPS> (+ 2 2.0)
4.0
CLIPS>
```

Notice that in the last case of mixed arguments, CLIPS converts the result to standard double-precision floating-point type.

You can explicitly convert one type to another by using the float and integer functions, as demonstrated in the following examples.

```
; convert integer
; to float
CLIPS> (float (+ 2 2))
4.0
; convert float
; to integer
CLIPS> (integer (+ 2.0 2.0))
4
CLIPS>
```

Parentheses are used to explicitly specify the order of expression evaluation if desired. In the example of ?x + ?y * ?z, the customary infix way to evaluate it is to multiply ?y by ?z and then add the result to ?x. However, in CLIPS, you must write the precedence explicitly if you want this order of evaluation, as follows.

```
(defrule mixed-calc
    (numbers ?x ?y ?z)
=>
    ; ?y * ?z + ?x
    (assert (answer (+ ?x (* ?y ?z)))))
```

In this rule, the expression in the innermost parentheses is evaluated first; so ?y is multiplied by ?z. The result is added to ?x.

Bound Bachelors

The analog to assigning a value to a variable on the LHS by pattern matching is **binding** a value to a variable on the RHS using the **bind function**. It's convenient to bind variables on the RHS if the same values will be repeatedly used.

As a simple example in a math calculation, let's first bind the answer to a variable and then print the **bound variable**.

```
CLIPS> (clear)
CLIPS>
(defrule addition
   (numbers ?x ?y)
=>
   (assert (answer (+ ?x ?y)))
   (bind ?answer (+ ?x ?y))
   (printout t "answer is "
              ?answer crlf))
CLIPS> (assert (numbers 2 2))
<Fact-1>
CLIPS> (run)
answer is 4
CLIPS> (facts)
f-1     (numbers 2 2)
f-2     (answer 4)
For a total of 2 facts.
CLIPS>
```

The (bind) also can be used on the RHS to bind single or multifield values to a variable. The (bind) is used to bind zero, one, or more values to a variable *without the "$" operator*. Recall that on the LHS, you can only create a multifield pattern by using the "$" operator on a field, such as "$?x". However, the "$" is unnecessary on the RHS because the arguments of (bind) explicitly tell CLIPS exactly how many values to bind. In fact, the "$" is a useless appendage on the RHS.

The following rule illustrates some variable bindings on the RHS. The **multifield value function**, **create$**, is used to create a multifield value. Its general syntax is as follows.

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

where any number of arguments can be appended together to create a multifield value. This multifield value, or a single-field value, can then be bound to a variable as shown in the RHS actions of the following rule.

```
CLIPS> (clear)
CLIPS>
(defrule bind-values-demo
```

```
       =>
          (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 ()
   CLIPS>
```

Doing Your Own Thing

Just like other languages, CLIPS allows you to define your own functions with **deffunction**. The deffunction is known globally, which saves you the effort of entering the same actions over and over again.

Deffunctions also help in readability. You can call a deffunction just like any other function. A deffunction may also be used as the argument of another function. A (printout) can be used *anywhere* in a deffunction even if it's not the last action because printing is a side-effect of calling the (printout) function.

The general syntax of a deffunction is shown following.

```
(deffunction <function-name>
   [optional comment]
   ; argument list. Last one may
   ; be optional multifield arg.
```

```
(?arg1 ?arg2 ... ?argM [$?argN])
; action1 to action(K-1)
; do not return a value
; only last action returned
(<action1>
 <action2>
   ...
 <action(K-1)>
 <actionK>)
```

The *?arg* are **dummy arguments**, which mean that the names of the arguments will not conflict with variable names in a rule if they are the same. The term dummy argument is sometimes called a **parameter** in other books.

Although each action may have returned values from function calls within the action, these are blocked by the deffunction from being returned to the user. The deffunction will only return the value of the *last* action, <actionK>. This action may be a function, a variable, or a constant.

The following is an example of how a deffunction is defined to calculate the hypotenuse, and then used in a rule. Even if the variable names in the rule are the same as the dummy arguments, there's no conflict. That's why they're *dummy*, because they don't mean anything.

```
CLIPS> (clear)
CLIPS>
(deffunction hypotenuse   ; name
   ; dummy arguments
   (?a ?b)
   ; action
   (sqrt(+ (* ?a ?a) (* ?b ?b))))
CLIPS>
(defrule calculate-hypotenuse
   (dimensions ?base ?height)
=>
   (printout t "Hypotenuse="
             (hypotenuse ?base ?height)
             crlf))
CLIPS> (assert (dimensions 3 4))
<Fact-1>
CLIPS> (run)
```

```
Hypotenuse=5.0
CLIPS>
```

Deffunctions may be used with multifield values, as the following example shows.

```
CLIPS> (clear)
CLIPS>
(deffunction count ($?arg)
    (length$ $?arg))
CLIPS> (count 1 2 3 a duck "quacks")
6
CLIPS>
```

Other Features

Other useful functions follow. For more information, see the *CLIPS Reference Manual*.

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 deffunction.

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 it exists, else nil.

member$: Number of the field if literal or variable exists, else FALSE.

subsetp: Returns TRUE if a multi field 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.

How to Be in Control

> *When you're young, you're controlled by the world, when you're older, you*
> *should control the world*

Up to this point, you've been learning the basic syntax of CLIPS. Now you'll see how to apply the syntax you've learned to more powerful and complex programs. You'll also learn some new syntax for input, and see how to compare values and generate loops.

Let's Start Reading

Besides matching a pattern, a rule can get information in another way. CLIPS can read the information that you type from the keyboard using the **read** function.

The following example shows how (read) is used to input data. Note that no extra (crlf) is needed after the (read) to put the cursor on a new line. The (read) automatically resets the cursor to a new line.

```
CLIPS> (clear)
CLIPS>
(defrule read-input
=>
   (printout t "Name a primary color"
               crlf)
   (assert (color (read))))
CLIPS>
(defrule check-input
   ?color <-
      (color ?color-read&red|yellow|blue)
=>
   (retract ?color)
   (printout t "Correct" crlf))
CLIPS> (reset)
CLIPS> (agenda)
0      read-input: *
```

```
For a total of 1 activation.
CLIPS> (run)
Name a primary color
red
Correct
CLIPS> (reset)
CLIPS> (run)
Name a primary color
green
CLIPS>    ; No "correct"
```

The rule is designed to use keyboard input on the RHS, so it's convenient to trigger the rule by not specifying any patterns on the LHS so it will automatically be activated when a (reset) occurs. When the activation for the read-input rule is displayed by the (agenda) command, an * is printed rather than a fact identifier such as f-1. The * is used to indicate that the pattern is satisfied, but not by a specific fact.

The (read) function is not a general-purpose function that will read anything you type on the keyboard. One limitation is that (read) will read only one field. So if you try to read

```
primary color is red
```

only the first field, "primary", will be read. To (read) all the input, you must enclose the input within double quotes. Of course, once the input is within double quotes, it is a single literal field. You can then access the substrings "primary", "color", "is", and "red" with the **str-explode** or **sub-string functions**.

The second limitation of (read) is that you can't input parentheses unless they are within double quotes. Just as you can't assert a fact containing parentheses, you can't (read) parentheses directly except as literals.

The **readline** function is used to read multiple values until terminated by a carriage return. This function reads in data as a string. In order to assert the (readline) data, an (assert-string) function is used to assert the nonstring fact, just as input by (readline). A top-level example of (assert-string) follows.

```
CLIPS> (clear)
CLIPS>
(assert-string "(primary color is red)")
<Fact-1>
CLIPS> (facts)
```

```
   f-1        (primary color is red)
   For a total of 1 fact.
   CLIPS>
```

Notice that the argument of (assert-string) must be a string The following shows how to assert a fact of multiple fields from (readline).

```
   CLIPS> (clear)
   CLIPS>
   (defrule test-readline
   =>
      (printout t "Enter input" crlf)
      (bind ?string (readline))
      (assert-string
         (str-cat "(" ?string ")")))
   CLIPS> (reset)
   CLIPS> (run)
   Enter input
   primary color is red
   CLIPS> (facts)
   f-1        (primary color is red)
   For a total of 1 fact.
   CLIPS>
```

Since (assert-string) requires parentheses around the string to be asserted, the (str-cat) function is used to put them around ?string.

Both (read) and (readline) also can be used to read information from a file by specifying the logical name of the file as the argument. For more information, see the *CLIPS Reference Manual*.

Being Efficient

CLIPS is a rule-based language that uses a very efficient pattern-matching algorithm called the **Rete Algorithm**, devised by Charles Forgy of Carnegie-Mellon University for his OPS shell. The term *Rete* is Latin for *net*, and describes the software architecture of the pattern-matching process.

It is very difficult to give precise rules that will always improve the efficiency of a program running under the Rete Algorithm. However, the following should be taken as general guidelines that may help:

1. Put the most specific patterns in a rule first. Patterns with unbound variables and wildcards should be lower down in the list of rule patterns. A control fact should be put first in the patterns.

2. Patterns with fewer matching facts should go first to minimize partial matches.

3. Patterns that are often retracted and asserted, **volatile patterns**, should be put last in the list of patterns.

As you can see, these guidelines are potentially contradictory. A non-specific pattern may have few matches (see guidelines 1 and 2). Where should it go? The overall guideline is to minimize changes of the partial matches from one cycle of the Inference Engine to the next. This may require much effort by the programmer in watching partial matches. An alternative solution is simply to buy a faster computer, or an accelerator board. This is becoming more attractive since the price of hardware always goes down while the price of human labor always goes up. Because CLIPS is designed for portability, any code developed on one machine should work on another.

Other Features

The **test conditional element** provides a very powerful way by which to compare numbers, variables, and strings on the LHS. The (test) is used as a pattern on the LHS. A rule will only be triggered if the (test) is satisfied together with other patterns.

Many predefined functions are provided by CLIPS. Logical functions are:

not: Boolean *not*

and: Boolean *and*

or: Boolean *or*

Arithmetic functions are:

/: Division

*: Multiplication

+: Addition

-: Subtraction

Comparison functions are:

eq: Equal (any type). Compares type and magnitude.

neq: Not equal (any type).

=: Equal (numeric type). Compares magnitude.

<>: Not equal (numeric type).

>=: Greater than or equal to

>: Greater than.

<=: Less than or equal to.

<: Less than.

All the comparison functions except "eq" and "neq" will give an error message if they are used to compare a number and non-number. If the type is not known in advance, the "eq" and "neq" functions should be used. The eq function checks for the same magnitude *and* type of its arguments while the "=" function *only* checks the magnitude of its (numeric) arguments and doesn't care if they're integer or floating-point.

The **logical functions** of CLIPS are **and**, **or**, and **not**. They can be used in expressions as Boolean functions. In CLIPS, true and false are represented by the symbols TRUE and FALSE. Note that upper-case *must* be used for logical values in CLIPS.

In addition to all the predefined functions, you may write **external functions** or **user-defined functions** in C, Ada, or other procedural languages and link to CLIPS. These external functions are then used as you would any predefined function.

CLIPS also gives you the capability of specifying an explicit **and conditional element**, an **or conditional element**, and a **not conditional element** on the LHS. The absence of a fact is specified as a pattern on the LHS using the "not" conditional element.

The alteration of our information to conform to reality is called **truth maintenance**. That is, we try to maintain the state of our minds to contain only true information so as to minimize conflicts with the real world.

While people can do this fairly easily (practice makes perfect), it's difficult for computers because they don't normally know which pattern entities are **logically dependent** on other pattern entities. CLIPS has a feature to support truth maintenance which will internally tag those pattern entities which are logically dependent on others. If these other pattern entities are retracted, CLIPS will automatically retract the logically dependent ones. The **logical conditional element** uses the keyword **logical** around a pattern to indicate that the matching pattern entities provide **logical support** to the assertions on the RHS.

Although the logical support works for assertions, it does *not reassert* retracted facts. The moral is, if you lose something due to erroneous information, you can't get it back (like losing money on your stockbrokers advice.)

CLIPS has two functions to help with logical support. The **dependencies** function lists the partial matches from which a pattern entity receives logical support, or none if there is no support. The second logic function is **dependents** which lists all pattern entities which receive logical support from a pattern entity.

The connective constraint, uses "&", "|", or "~". Another type of field constraint is called a **predicate constraint** and is often used for pattern matching of more complex fields. The purpose of a predicate constraint is to constrain a field depending on the result of a Boolean expression. If the Boolean returns FALSE, the constraint is not satisfied and the pattern matching fails. You'll find that the predicate constraint is very useful with numeric patterns.

A **predicate function** is one which returns a FALSE or a non-FALSE value. The **colon**, "**:**" followed by a predicate function is called a **predicate constraint**. The ":" may be preceded by "&", "|", or "~" or may stand by itself as in the pattern (fact :(> 2 1)). It is typically used with the & connective constraint as "**&:**". The following list contains some of the predicate functions defined by CLIPS:

(**evenp** <arg>): Check if <arg> is even number.

(**floatp** <arg>): Check if <arg> is floating-point number.

(**integerp** <arg>):   Check if <arg> is integer.

(**lexemep** <arg>): Check if <arg> is symbol or string.

(**multifieldp** <arg>): Check if <arg> is multifield value.

(**numberp** <arg>): Check if <arg> is float or integer.

(**oddp** <arg>): Check if <arg> is odd number.

(**pointerp** <arg>): Check if <arg> is external address.

(**stringp** <arg>): Check if <arg> is string.

(**symbolp** <arg>): Check if <arg> is symbol.

There are often cases in which it's convenient to have values which are globally known in an expert system. For example, it is inefficient to have to redefine universal constants such as π. CLIPS provides the **defglobal** construct so that values may be universally known to all rules.

Another type of useful function is random numbers. CLIPS has a **random** function which returns a "random" integer value. The random number function of CLIPS actually returns **pseudorandom** numbers, which means they are not truly random but are generated by a mathematical formula. For most purposes the pseudorandom numbers will be fine. Note that the random function of CLIPS uses the ANSI C library function rand which may not be available on

all computers that do not adhere to this standard. For more information on all these topics, please see the *CLIPS Reference Manual.*

In addition to control facts to control the execution of programs, CLIPS provides a more direct way of control by the explicit assignment of salience to rules. The main problem associated with explicitly using salience while you were just starting to learn CLIPS is the tendency to overuse salience and write sequential programs. This overuse defeats the whole purpose of using a rule-based language, which is to provide a natural vehicle for those applications best represented by rules. In the same way, procedural languages are best for strong control-oriented applications, while object-oriented languages are best for representing objects. CLIPS has keywords called **declare salience** which can be used to explicitly set the priority of rules.

Salience is set using a numeric value ranging from the smallest value of -10000 to the highest of 10000. If a rule has no salience explicitly assigned by the programmer, CLIPS assumes a salience of zero. Notice that a salience of zero is midway between the largest and smallest salience values. A salience of zero does not mean that the rule has no salience but, rather, that it has an intermediate priority level.

CLIPS provides some procedural programming structures that can be used on the RHS. These structures are the **while** and **if then else** that also are found in modern high-level languages such as Ada, C, and Java.

Another useful function with (while) loops is **break** which ends the currently executing (while) loop. The **return** function immediately ends the currently executing deffunction, generic function, method, or message-handler.

Any function may be called from the RHS, which greatly contributes to the power of CLIPS. Many other CLIPS functions are available that may return with numbers, symbols, or strings. These functions may be used for their return values or for their **side-effects**. An example of a function only used for its side-effect is (printout). The value returned by (printout) is meaningless. The importance of (printout) is in its side-effect of output. In general, functions may have nested arguments if appropriate to your desired effect.

Before a file can be accessed for reading or writing, it must be opened using the **open** function. The number of files that can be opened at once is dependent on your operating system and hardware. When you no longer need to access a file, you should close it with the **close** function. Unless a file is closed, there is no guarantee that the information written to it will be saved.

The **logical name** of a file is how CLIPS identifies the file. The logical name is a global name by which CLIPS knows this file in *all* rules. Although the logical name could be identical to

the filename, you may want to use something different. Another advantage of a logical name is that you can easily substitute a different filename without making major program changes.

The function to read data from a file is the familiar (read) or (readline). The only new thing that you have to do is to specify the logical name from which to read as the argument of (read) or (readline).

To (read) more than one field, you must use a loop. Even with (readline), a loop is necessary to read multiple lines. A loop can be written by having one rule trigger another or with a while-loop. The loop should not try to read past the end of file or the operating system will issue an error message. To help prevent this, CLIPS returns an **EOF** symbolic field if you try to read past the end of file (EOF).

The **evaluation** function, **eval**, is used for evaluating any string or symbol except the "def" type constructs such as defrule, deffacts, etc., as if entered at the top-level. The **build** function takes care of the "def" type constructs. The (build) function is the complement of (eval). The build function evaluates a string or symbol as if it were entered at the top-level and returns TRUE if the argument is a legal def-type construct such as (defrule), (deffacts), and so forth.

Chapter 8

Matters of Inheritance

The easiest way to obtain wealth is to inherit it; the second best way is to make it off the labor of others; marrying wealth is too much like work.

This chapter is an overview of object-oriented programming in CLIPS. Unlike rule-based programming in which you can just jump right in and write a rule without caring what else is in the system, object-oriented programming requires some essential background material.

How to be Objective

A key characteristic of good program design is *flexibility*. Unfortunately, the rigid methodology of structured programming techniques does not provide the needed flexibility for fast, reliable, and efficient changes. The **object-oriented programming** (**OOP**) **paradigm**. provides this flexibility.

The term *paradigm* comes from the Greek word *paradeigma* which means a model, example, or pattern. In computer science, a *paradigm* is a consistent, organized methodology for trying to solve a problem. Today, there are many programming paradigms such as OOP, **procedural**, **rule-based**, and **connectionist**. The term **artificial neural systems**, is a modern synonym for the older term *connectionist*.

Traditional programming is procedural because it emphasizes algorithms or procedures in solving problems. Many languages have been developed to support this procedural paradigm, such as Pascal, C, Ada, FORTRAN, and BASIC. These languages have also been adapted for **object-oriented design** (**OOD**) by either adding extensions or imposing a design methodology on the programmers. In contrast, new languages have been developed to provide OOP, which is *not* the same as OOD. You can do OOD in any language, even assembly language.

CLIPS provides three paradigms: rules, objects, and procedures. You will learn more about the objects in the **CLIPS Object-Oriented Language** (**COOL**) which is integrated with the rule and procedural based paradigms of CLIPS. CLIPS supports the procedural paradigm through generic functions, deffunctions, and user-defined external functions. Depending on the application, you can use rules, objects, procedures, or a combination.

Rather than imposing a single paradigm on the user, our philosophy is that a variety of specialized tools, a **multi-paradigm** approach, is better than trying to force everyone to use a single general purpose tool. As an analogy, while you could use a hammer and nails for fastening *everything*, there are cases in which other fasteners are preferred. For example, imagine fastening your pants with a hammer and nails instead of a zipper. (NOTE: if anyone does use a hammer and nails on their pants, please contact the Guiness Book of World Records.)

The Class Stuff

In OOP a class is a **template** which describes the common characteristics or **attributes** of objects. Note that this use of the term template is not the same as a *deftemplate* as described in an earlier chapter. Here, the word template is used in the sense of a tool that is used to build objects having common attributes. As analogies, a straightedge is a template for drawing straight lines while a cookie-cutter is a curvaceous template.

Classes of objects are arranged in a hierarchy or in a graph to describe the relationships of objects in a system. Each class is an **abstraction** of a real-world system or some other logical system that we are trying to model. For example, one abstract model of a real-world system might be an automobile. Another abstract model of a logical system could be financial instruments such as stocks and bonds, or complex numbers. The term *abstraction* refers to (1) the abstract description of a real-world object or other system that we are trying to model, or (2) the process of representing a system in terms of classes. Abstraction is one of the five generally accepted features of a true OOP language. The others are **inheritance**, **encapsulation**, **polymorphism** and **dynamic binding**. These terms will be explained in detail as you read through this book. CLIPS supports all five of these features.

The term *abstract* means that we are not concerned with nonessential details. An abstract description of a complex system is a simplified description that concentrates on relevant information for a specific purpose. Thus, the system is represented by a simpler, easier to understand model. As a familiar example, when certain people drive cars, they utilize an abstract model of driving that consists of two items — the steering wheel and the accelerator. That is, these people are not concerned with the hundreds of components that make up an automobile, nor the theory of internal combustion engines, traffic laws, and so forth. Knowing only how to use the steering wheel and accelerator is their abstract model of driving.

One of the five fundamental features of OOP is inheritance. Classes are arranged in a hierarchy with the most general classes at the top and the more specialized classes below. This allows new classes to be easily defined as specialized refinements or modifications of existing classes.

The use of inheritance can greatly speed up software development and increase reliability because new software does not need to be created from scratch each time a new program is made. OOP makes it easy to utilize **reusable code**. OOP programmers often make use of object libraries consisting of hundreds or thousands of objects. These objects can be used or modified as desired in a new program. In addition to public domain object libraries, a number of companies market commercial object libraries. Although the concept of reusable software components has been around since the early days of FORTRAN subroutine libraries in the 1960s, the concept has never before been so successfully used for general software development.

In order to define a class, you must specify one or more **parent classes** or **superclasses** of the class to be defined. As an analogy to superclasses, every person has parents; people do not spontaneously come into existence (although sometimes you may wonder if certain people really had parents.) The opposite of a superclass is a **child class** or **subclass**.

This determines the inheritance of the new class. A subclass inherits **attributes** from one or more superclasses. The term *attribute* in COOL refers to the **properties** of an object, which are named **slots** that describe it. For example, an object to represent a person might have slots for name, age, address, and so forth.

An **instance** is an object that has *values* for the slots such as John Smith, 28, 1000 Main St., Clear Lake City, TX. Lower-level classes automatically inherit their slots from higher-level classes, unless the slots are explicitly blocked. New slots are defined in addition to the inherited slots to set all the attributes that describe the class.

An object's **behavior** is defined by its **message-handlers**, or **handlers** for short. A message-handler for an object responds to **messages** and performs the required actions. For example, sending the message

```
(send [John_Smith] print)
```

would cause the appropriate message-handler to print the values of the slots of the instance John_Smith. Instances are generally specified within **brackets**, **[ ]**. A message begins with the **send** function, followed by the instance name, message name, and any required arguments. For example, in the case of the print message, there are no arguments. An **object** in CLIPS is an instance of a class.

The encapsulation of slots and handlers inside an object is another of the five generally accepted features of an OOP. The term *encapsulated* means that a class is defined in terms of its slots and handlers. Although an object of a class may inherit slots and handlers from its superclasses, with a few exceptions discussed later, *the object's slot values cannot be altered or examined without sending a message to the object.*

The **root class** or simply **root** of CLIPS is a **predefined system class** called **OBJECT**. The predefined system class **USER** is a subclass of OBJECT.

How the UPPIES Get Theirs

As an example, suppose we wanted to define a class called UPPIE, which is a colloquial term for **urban professional**. Note that in this book, we'll follow the convention of writing classes in all uppercase.

Fig. 8.1 illustrates how the UPPIEs get their inheritance all the way up to the root class OBJECT. Notice that UPPIE is defined as a subclass of USER. The boxes or **nodes** represent classes while the connecting arrows are called **links**. Lines are often used instead of arrows for simplicity in drawing. Also, since CLIPS supports only **is-a** links, the "is-a" relationship will not be explicitly written next to each link from now on.

Fig. 8.1 The UPPIE Class
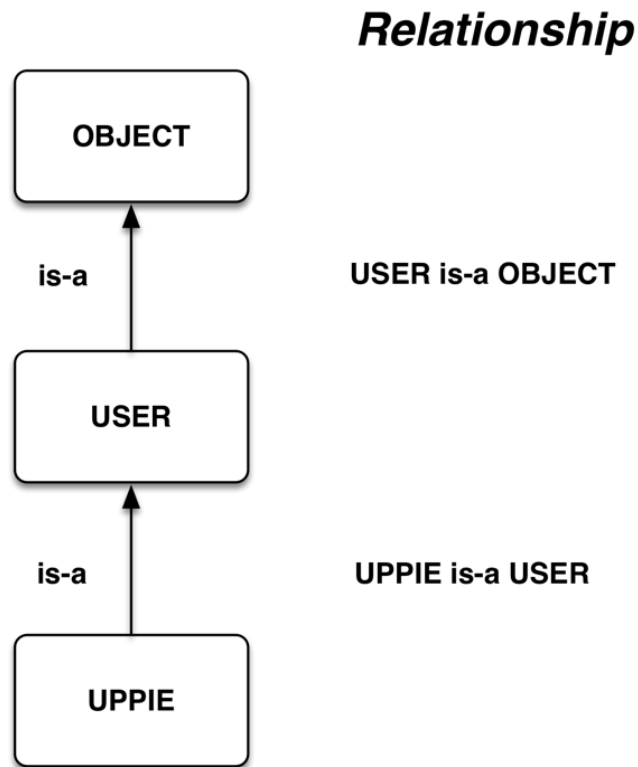
The convention we will follow for the **relationship** between classes is that the tail end of the arrow is on the subclass while the head points at the superclass. The relationships in Fig. 8.1 follow this convention. Another possible convention is to use arrows to point at the subclasses.

The is-a link indicates the inheritance of slots from a class to its subclass. A class may have zero or more subclasses. All classes except OBJECT must have a superclass. Since UPPIE also inherits all slots of USER, and USER inherits all slots of OBJECT, it follows that UPPIE inherits all slots of OBJECT too. The same principle of inheritance also applies to the message-handlers of each class. For example, UPPIE inherits all the handlers of USER and OBJECT.

The inheritance of slots and handlers is particularly important in OOP since it means that you do not have to redefine the properties and behavior of each new class of objects that is defined. Instead, each new class inherits all the properties and behavior from its higher-level classes. Since the new behavior is inherited, it may substantially reduce the **verification and validation** (**V&V**) of the handlers. V&V essentially means that the product was built properly and that it meets the requirements. The task of verifying and validating software may take more time and money than the software development itself, especially if the software affects human life and property. Inheritance of handlers allows for efficient reuse of existing code and speeds up development.

Classes are defined in CLIPS using the **defclass** construct. The UPPIE class is defined in one statement as follows.

```
(defclass UPPIE (is-a USER))
```

Notice the similarity between the UPPIE–USER relationship in Fig. 8.1 and the (defclass) construct.

You do not have to enter the USER or the OBJECT classes since these are predefined classes and so CLIPS already knows their relationship. In fact, if you try to define USER or OBJECT, an error message will result since you cannot change the predefined classes, unless you change the source code of CLIPS.

Since CLIPS is **case-sensitive**, commands and functions *must* be entered in lowercase. Predefined system classes such as USER and OBJECT *must* be entered in uppercase. Although you can enter user-defined classes in lowercase or uppercase, we will follow the convention of using all uppercase for classes for the sake of readability.

The basic format of the defclass command to define classes only, and not slots, is,

```
(defclass <class>
    (is-a <direct-superclasses>))
```

The list of classes, <direct-superclasses>, is called a **direct superclass precedence list** because it defines how a class is linked to its direct superclasses. The direct superclasses of a class are the one or more classes named after the is-a keyword. In our example, class DUCKLING is

the direct superclass of DUCK. Note that at least *one* direct superclass must be given in the direct superclass precedence list.

If the direct superclass list was as follows,

```
(defclass DUCK
    (is-a DUCKLING USER OBJECT))
```

then USER and OBJECT would also be direct superclasses of DUCK. In this example, it makes no difference whether USER and OBJECT are specified in addition to DUCKLING. In fact, since USER and OBJECT are predefined classes that are always linked such that USER is-a OBJECT, and OBJECT is the root, you need never specify them except when defining a subclass of USER. Since USER only inherits from OBJECT, it is not necessary to specify OBJECT if USER is specified.

The **indirect superclasses** of a class are all the classes *not* named after the "is-a" that contribute slots and message-handlers by inheritance. In our example, the indirect superclasses are USER and OBJECT. A class inherits slots and message-handlers from *all* its direct and indirect superclasses. Thus, DUCK inherits from DUCKLING, USER, and OBJECT.

A **direct subclass** is connected by a *single* link to the class above it. An **indirect subclass** has *more than one link*. Fig. 8.2 summarizes the class terminology.

Fig. 8.2 Class Relationships
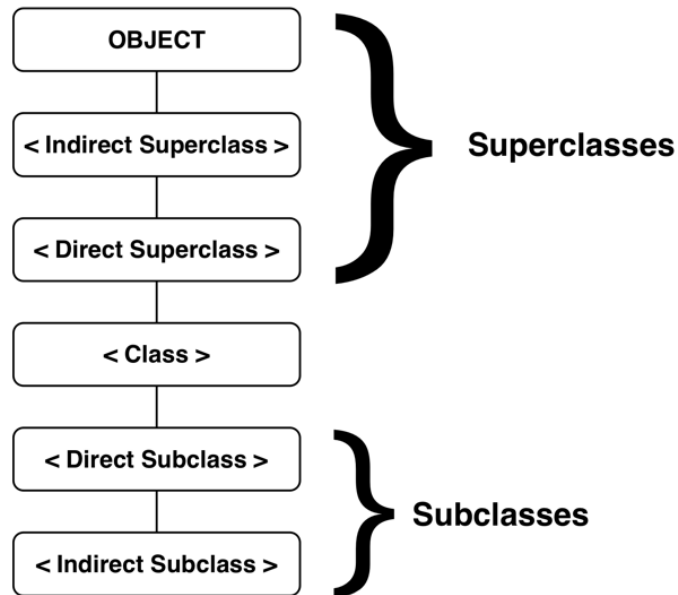
The root class OBJECT is the *only* class that does not have a superclass.

Using this fancy new terminology allows us to state the *Principle of OOP Inheritance*: A class may inherit from all its superclasses.

This is a simple, yet powerful concept fully exploited in OOP. This principle means that slots and message-handlers may be inherited to save us the trouble of redefining them for new subclasses. In addition, slots may be easily customized for new subclasses as modifications and as composites of superclass slots. By allowing easy and flexible reuse of existing code, program development time and cost are decreased. In addition, the reuse of working, existing code minimizes the amount of verification and validation needed. All these advantages facilitate the program maintenance tasks of debugging, modification, and enhancement once the code is released.

The reason for using *may* in the principle is to emphasize that inheritance of slots from a class may be blocked by including a no-inherit facet in the class slot definition.

The direct and indirect classes of a class are all those that lie on an **inheritance path** to OBJECT. An *inheritance path* is a set of connected nodes between the class and OBJECT. In our example, the single inheritance path of DUCK is DUCK, DUCKLING, USER, and OBJECT. You will see examples later, such as Fig. 8.5, in which a class has **multiple inheritance** paths to OBJECT.

Fig. 8.3 illustrates a very simplified **taxonomy** of organisms that illustrates inheritance in Nature. The term *taxonomy* means a classification. Biological taxonomies are designed to show the kinship of organisms. That is, a biological taxonomy emphasizes similarities between organisms by grouping them together.

Fig. 8.3 Simple Taxonomy of Living Organisms With is-a Links
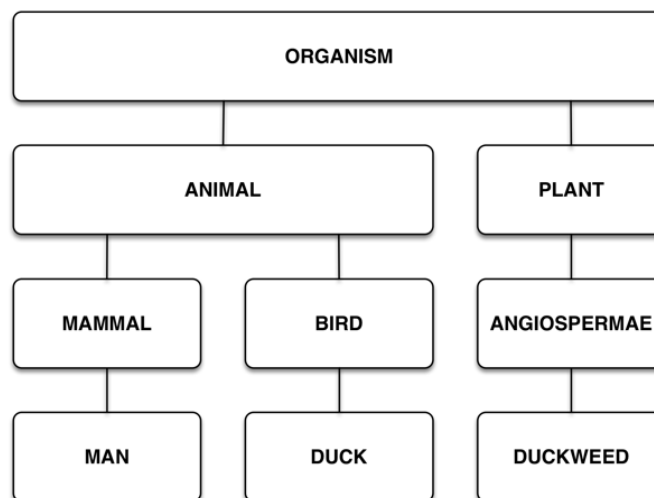
In a taxonomy like Fig. 8.3, the connecting lines are *all* is-a links. For example, a DUCK is-a BIRD. A BIRD is-a ANIMAL. An ANIMAL is-a ORGANISM and so forth. Although the genetic inheritance of each *individual* is different, the characteristics of MAN and DUCK are the same for each species.

In Fig. 8.3, notice that the most general class, ORGANISM, is at the top, while more specialized classes are lower in the taxonomy. In CLIPS terminology, we would say that each subclass inherits the *slots* of its parent classes. For example, since mammals are warm-blooded and give birth to live young, with the exception of the platypus, the class MAN inherits the attributes of the parent MAMMAL class. The direct superclass of MAMMAL is ANIMAL and the direct subclass of MAMMAL is MAN. The indirect superclass of MAMMAL is ORGANISM.
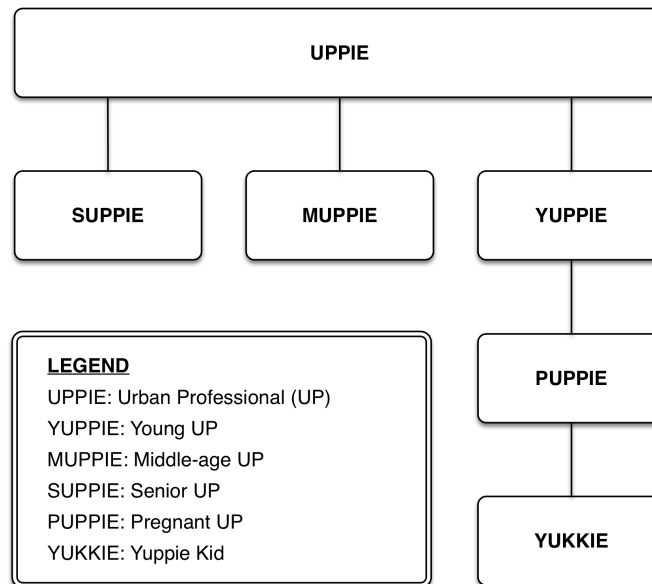
The other classes such as BIRD, DUCK, and so forth have no relationship to MAMMAL because they are not on an **inheritance path** from the most general class ORGANISM. An inheritance path is any path from one class to another that does not involve **backtracking** or retracing the path. A class such as PLANT is not on an inheritance path to MAMMAL because we would have to backtrack to ORGANISM before continuing down to MAMMAL. Thus, MAMMAL does not automatically obtain any slots from PLANT or any other classes not on the inheritance path to MAMMAL. This model of inheritance mirrors the real world, since otherwise we might have grass growing on our heads instead of hair.

The Illegitimate YUKKIE

Now that you've got the basic idea of classes, let's add some additional classes to the UPPIE diagram of Fig. 8.1 in order to make the example more realistic. This type of development by adding lower-level classes is the way that OOP is done, by adding classes from the most general to the most specific classes.

Fig. 8.4 shows the inheritance diagram of the illegitimate YUKKIE**.** For simplicity, the OBJECT and USER classes are not shown. The hierarchy of Fig. 8.4 is a **tree** because every node has exactly one parent.

Fig. 8.4 The Illegitimate YUKKIE

UPPIE

SUPPIE   MUPPIE   YUPPIE

PUPPIE

YUKKIE

LEGEND
UPPIE: Urban Professional (UP)
YUPPIE: Young UP
MUPPIE: Middle-age UP
SUPPIE: Senior UP
PUPPIE: Pregnant UP
YUKKIE: Yuppie Kid

A familiar example of a tree organizational structure is that often used by companies which have a **hierarchy** consisting of president, vice-presidents, department heads, managers and so forth down to the lowest employee. In this case, the hierarchical structure mirrors the authority of people in the organization. Trees are generally used for organizations of people because every person has exactly one boss, except the top dog who has no boss. Nodes in the organizational chart represent the positions such as president, vice president, etc. Lines connecting the positions are the **branches** that indicate the division of responsibility. Links are often called branches in a tree.

In Fig. 8.4, every class except YUKKIE is legal or **legitimate**. For example, a SUPPIE is-a UPPIE. A MUPPIE is-a UPPIE. A YUPPIE is-a UPPIE. A PUPPIE is-a YUPPIE (no middle-aged YUPPIE moms allowed.) We would also like to say that a YUKKIE is-a YUPPIE and a YUKKIE is-a UPPIE by inheritance. However, we *don't* want to say that a YUKKIE is-a PUPPIE, which is what the is-a link between YUKKIE and PUPPIE means.

The is-a link between YUKKIE and PUPPIE is a natural mistake for a person to make since a YUKKIE is the child of a PUPPIE (actually an ex-PUPPIE after she gives birth.) Although making an is-a link between YUKKIE and PUPPIE does allow the YUKKIE to inherit from YUPPIE and UPPIE as desired, it also produces an illegitimate relationship by saying that a YUKKIE is-a PUPPIE. This means that a YUKKIE will inherit all the slots of a PUPPIE. Assuming that one of the PUPPIE slots specifies how many months pregnant the PUPPIE is, this means that every Yuppie kid will have a slot to indicate how many months pregnant he or she is too!

It is possible to correct the figure. However, we need to use a graph instead of a tree. In contrast to trees, in which every node except the root has exactly one parent, each node in a graph may have zero or more nodes connected to them. A familiar example of a graph is a roadmap in which cities are nodes, and roads are the links connecting them. Another difference between trees and graphs is that most types of trees have a hierarchical structure while general types of graphs do not.

Fig. 8.5 shows the legitimate Yuppie class YUKKIE. A new class CHILD has been created and is-a links made between YUKKIE and its two superclasses, YUPPIE and CHILD. Notice there is no longer an illegitimate link between YUKKIE and PUPPIE.

Fig. 8.5 The Legitimate YUKKIE
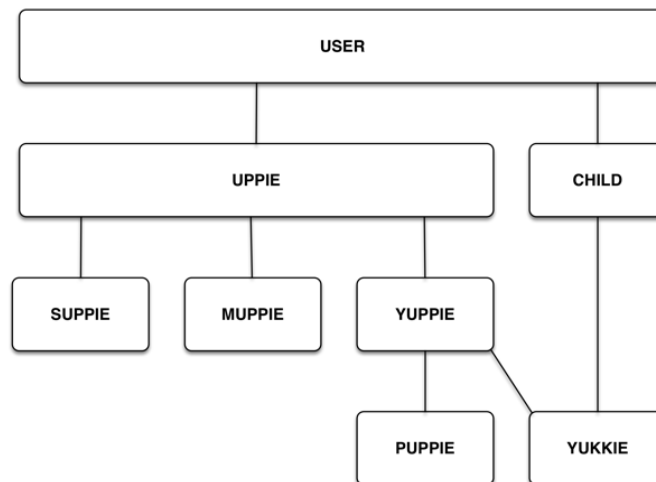
This is a graph because the YUKKIE class has two direct superclasses instead of only one as in a tree. This is also a hierarchical graph because the classes are arranged using is-a links from the most general, USER, to most specific, SUPPIE, MUPPIE, PUPPIE, and YUKKIE. Using Fig. 8.5, we can say that a YUKKIE is-a YUPPIE, and also that a YUKKIE is-a CHILD.

Shown following are the commands to add the subclasses shown in Fig. 8.5.

```
CLIPS> (clear)
CLIPS> (defclass UPPIE (is-a USER))
CLIPS> (defclass CHILD (is-a USER))
CLIPS> (defclass SUPPIE (is-a UPPIE))
CLIPS> (defclass MUPPIE (is-a UPPIE))
CLIPS> (defclass YUPPIE (is-a UPPIE))
CLIPS> (defclass PUPPIE (is-a YUPPIE))
CLIPS> (defclass YUKKIE (is-a YUPPIE CHILD))
```

The order that classes are defined must be such that a class is defined before its subclasses. Thus,

```
(defclass CHILD
    (is-a USER))
```

must be entered *before*

```
(defclass YUKKIE
    (is-a YUPPIE CHILD))
```

CLIPS will issue an error message if you try to enter the YUKKIE class before the CHILD class.

Notice the left-to-right order that SUPPIE, MUPPIE, and YUPPIE are drawn in Fig. 8.5. This corresponds to the order that these classes are entered into CLIPS and is the convention that we will follow. You can also see why CHILD is drawn to the right of UPPIE since it was entered *after* the UPPIE class.

In Fig. 8.5, notice that the link YUKKIE—YUPPIE is drawn to the left of the link YUKKIE—CHILD. Another convention that we will follow is to write the direct superclasses from left to right in the precedence list according to their left-to-right order drawn in a graph. The ordering YUPPIE CHILD in the precedence list of YUKKIE is done to satisfy this convention.

Show Me

CLIPS provides a number of functions to show information about classes, such as predicate functions to test whether a class is a superclass or subclass of another.

The **superclassp** function returns TRUE if <class1> is a superclass of <class2>, and FALSE otherwise. The **subclassp** function returns TRUE if <class1> is a subclass of <class2>, and FALSE otherwise. The general form of both functions is

```
(function <class1> <class2>)
```

For example,

```
CLIPS> (superclassp UPPIE YUPPIE)
TRUE
CLIPS> (superclassp YUPPIE UPPIE)
FALSE
CLIPS> (subclassp YUPPIE UPPIE)
TRUE
CLIPS> (subclassp UPPIE YUPPIE)
```

```
    FALSE
    CLIPS>
```

Now let's check to see if CLIPS accepted all these new classes. One way of doing this is with the **list-defclasses** command. Following is the output of the command.

```
    CLIPS> (list-defclasses)
    FLOAT
    INTEGER
    SYMBOL
    STRING
    MULTIFIELD
    EXTERNAL-ADDRESS
    FACT-ADDRESS
    INSTANCE-ADDRESS
    INSTANCE-NAME
    OBJECT
    PRIMITIVE
    NUMBER
    LEXEME
    ADDRESS
    INSTANCE
    USER
    UPPIE
    CHILD
    SUPPIE
    MUPPIE
    YUPPIE
    PUPPIE
    YUKKIE
    For a total of 23 defclasses.
    CLIPS>
```

Notice that the (list-defclasses) command does not indicate the hierarchical class structure. That is, list-defclasses does not indicate which classes are subclasses or superclasses of others.

If you look down this list, you will see all the user-defined classes that you entered: UPPIE, CHILD, YUPPIE, MUPPIE, SUPPIE, PUPPIE, and YUKKIE. In addition to the predefined system classes OBJECT and USER that have been discussed so far, there are a number of other predefined classes. You should recognize most of them as having the same name as the familiar

CLIPS types that you learned about in earlier chapters. The predefined types of CLIPS are also defined as classes so that they can be used with COOL.

The general inheritance diagram of the predefined classes from the *CLIPS Reference Manual*, is shown in Fig. 8.6, where the arrows point to the subclasses.

Fig. 8.6 The Predefined CLIPS Classes
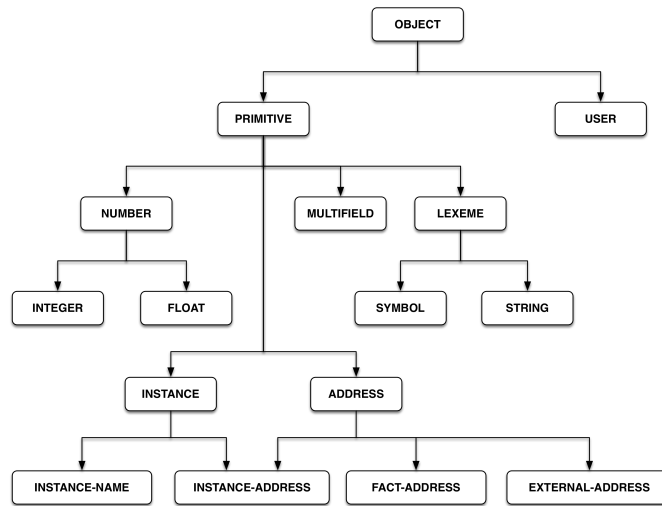
The OBJECT class is the root of the tree and is connected by **branches** to its subclasses. The terms *branch*, **edge**, *link*, and **arc** are basically synonymous in that they all indicate a connection between nodes. Each subclass is below its superclass(es).

Each subclass has higher **specificity** than its superclass. The term *specificity* means that a class is more restrictive. For example, LEXEME is a superclass of SYMBOL and STRING. If you were told that an object was of the LEXEME class, you would know that it could only be a SYMBOL or a STRING. However, if an object is a SYMBOL, it cannot be a STRING and *vice versa*. Thus, the classes SYMBOL and STRING are more *specific* than LEXEME.

The **browse-classes** command shows the class hierarchy through indentation.

```
CLIPS> (browse-classes)
OBJECT
  PRIMITIVE
    NUMBER
      INTEGER
      FLOAT
    LEXEME
      SYMBOL
      STRING
```

```
            MULTIFIELD
            ADDRESS
               EXTERNAL-ADDRESS
               FACT-ADDRESS
               INSTANCE-ADDRESS *
            INSTANCE
               INSTANCE-ADDRESS *
               INSTANCE-NAME
      USER
         UPPIE
            SUPPIE
            MUPPIE
            YUPPIE
               PUPPIE
               YUKKIE *
         CHILD
            YUKKIE *
CLIPS>
```

The asterisk after a class name indicates that it has multiple superclasses.

The (browse-classes) command has an optional argument which specifies the starting class for the subclasses you want to see. This is convenient if you are not interested in listing all the classes. The following examples illustrate how to show portions of the YUPPIE graph of Fig. 8.5, called **subtrees** or **subgraphs** depending on whether the nodes and links form a tree or graph.

```
CLIPS> (browse-classes UPPIE)
UPPIE
  SUPPIE
  MUPPIE
  YUPPIE
    PUPPIE
    YUKKIE *
CLIPS> (browse-classes YUPPIE)
YUPPIE
  PUPPIE
  YUKKIE *
CLIPS> (browse-classes YUKKIE)
```

```
YUKKIE *
CLIPS>
```

An **abstract class** is designed for inheritance only. The abstract class USER cannot have **direct instances** defined for it, which are instances defined directly for a class. In addition to the class information, the **inheritance precedence** of the classes is described. This is an **ordered list** in which the order from left to right indicates the highest to lowest precedence that classes contribute by inheritance. The inheritance precedence lists all the superclasses of a class back to the root class OBJECT. You can also see that the direct superclasses information indicates the superclass that is one link above a class while the inheritance precedence list shows *all* superclasses.

Even if a class has no direct instances, it will have **indirect instances** if it has subclasses which have instances. The indirect instances of a class are all the instances of its subclasses

A **concrete class** is allowed to have direct instances. For example, given a concrete class COW, the direct instance Elsie would be the famous TV salescow. Normally classes inheriting from abstract classes are also abstract classes. However, classes inheriting from system classes, such as USER, are considered to be concrete unless otherwise specified. So UPPIE and YUPPIE are also concrete classes.

It is strongly recommended that all classes you define in CLIPS be subclasses of USER. CLIPS will automatically provide handlers for print, init, and delete if your classes are subclasses of USER.

The YUKKIE class has **multiple inheritance** since it has two direct superclasses, CHILD and YUPPIE. If you think back to the analogy of inheritance to bosses in an organization, the issue of multiple inheritance brings up an interesting question— Who's the boss? In the case of a tree structure, every class has only one direct superclass (boss) and so its easy to figure out who to take orders from. However, in the case of multiple inheritance, there appears to be multiple "bosses" of equal authority which are the direct superclasses.

For the case of classes arranged in a tree, i.e., single inheritance, the inheritance is simply all classes along the inheritance path back to OBJECT. The inheritance path in a tree is just the shortest path from a class back to OBJECT. This concept of inheritance also applies to a subgraph of a graph which is a tree. For example, the inheritance precedence of the subgraph UPPIE, SUPPIE, MUPPIE, YUPPIE, and PUPPIE is a tree since each of them has only one parent. Thus, the inheritance precedence of each of them is the shortest sequence of links back to OBJECT. For example, the inheritance precedence of PUPPIE is PUPPIE, YUPPIE, UPPIE, USER, and OBJECT.

Fig. 8.7 is another example of a graph. Notice that some nodes such as rhombus have more than one parent.

Fig. 8.7 The Quadrilateral Graph
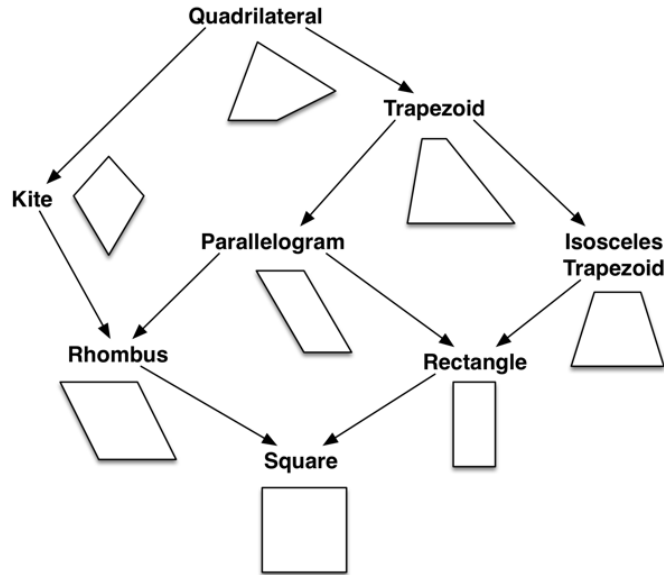
For simplicity, we will only discuss single inheritance in this book. For more details on multiple inheritance, see the *CLIPS Reference Manual*.

Other Features

Some other functions useful with classes are shown following.

ppdefclass: Pretty-print the declass internal structure.

undefclass: Eliminate class.

describe-class: Additional information about classes.

class-abstractp: Predicate function returns TRUE if the class is abstract.

For more information on these functions and topics mentioned in this chapter, see the *CLIPS Reference Manual*.

Chapter 9

Meaningful Messages

It's always better to please your superiors than your subordinates.
> — *Bowen*

In this chapter you'll learn more about classes and objects called instances. You will see how to specify the attributes of classes using slots and how to send messages to objects.

The Birds and the Bees

In Chapter 1 you learned the basic ideas of inheritance. The reason that inheritance is so important in OOP is that inheritance allows the easy construction of **customized software**. By *customized software*, we don't mean that the software is built from scratch. Rather, it's more like taking a mass produced item and modifying it for a special application. The mass produced item can be considered a product of a **software factory** that quickly, economically, and reliably produces items of a general type that are meant to be easily customized.

The heart of the OOP paradigm is the creation of a class hierarchy to quickly, easily, and reliably produce software. Generally, this software is a modification of existing software so that programmers are not always "reinventing the loop."

Although in the past, people have tried to provide reusable code through such mechanisms as subroutine libraries, the **pure OOP** paradigm in languages such as Smalltalk carries the concept of reusable code to its logical conclusion by trying to build *all* the software in a system as reusable code. In Smalltalk, everything is an object, even the classes. In CLIPS, instances of the primitive types such as NUMBER, SYMBOL, STRING and so forth, as well as user-defined instances are objects. Classes are *not* objects in CLIPS. For example, the NUMBER 1, the SYMBOL Duck, the STRING "Duck", and the user-defined instance Duck are all objects.

The OOP paradigm is quite different from the subroutine library approach in which bits and pieces of subroutine code may or may not be used depending on the programmer's whim. The OOP paradigm encourages and supports modular code— the message-handlers— that can be easily modified and maintained. This feature of code maintainability is playing an increasingly important role as the size and cost of systems increase.

A class in OOP is like a software factory which has the design information about an object. In other words, a class is like a template which can be used to produce identical objects that are the instances of the class. The classic analogy is that a class is like the blueprint of a cow, and the object that produces milk, such as Elsie, is the instance.

The general syntax of an instance name is simply a symbol surrounded by brackets, [ ], as follows.

```
[<name>]
```

The brackets are *not* actually part of the instance name, which is a symbol, like Elsie. Brackets are used to surround an instance name if there is danger of ambiguity in using the name. This may occur in the (send) function and so brackets are used in a (send). In case of doubt, use brackets, since it doesn't hurt.

Some examples of different types of objects in CLIPS are shown following:

SYMBOL

```
Dorky_Duck
```

STRING

```
"Dorky_Duck"
```

FLOAT

```
1.0
```

INTEGER

```
1
```

MULTIFIELD

```
(1 1.0 Dorky_Duck "Dorky_Duck")
```

DUCK

```
[Dorky_Duck]
```

The classes of **SYMBOL**, **STRING**, **FLOAT**, **INTEGER**, and **MULTIFIELD** have the same names as those you are familiar with from rule-based programming in CLIPS. These are called **primitive object types** because they are provided by CLIPS and automatically maintained as needed. These primitive types are mainly provided for use in **generic functions**. Two **compound classes** are **NUMBER** which is FLOAT or INTEGER, and **LEXEME** which is SYMBOL or STRING. The compound classes are provided for convenience if the type of number, or type of characters doesn't matter.

In contrast, **user-defined object types** are those that you define through user-defined classes. If you refer back to the predefined CLIPS classes of Fig. 8.6 in chapter 8, you'll see that primitive and user-defined classes are the top-level division of classes in CLIPS.

Two functions convert a symbol to an instance name, and *vice versa*. The **symbol-to-instance-name** converts a symbol to an instance name, as shown by the following.

```
; Get rid of any old classes
CLIPS> (clear)
CLIPS>
(symbol-to-instance-name Dorky_Duck)
[Dorky_Duck]
CLIPS>
(symbol-to-instance-name
    (sym-cat Dorky "_" Duck))
[Dorky_Duck]
CLIPS>
```

Notice how standard CLIPS functions such as (sym-cat), which concatenates items, can be used with the object system of CLIPS.

The opposite function, **instance-name-to-symbol**, converts an instance name to a symbol, as the following examples show.

```
CLIPS>
(instance-name-to-symbol [Dorky_Duck])
Dorky_Duck
CLIPS>
(str-cat
    (instance-name-to-symbol [Dorky_Duck])
    " is a DUCK")
"Dorky_Duck is a DUCK"
CLIPS>
```

Dorky Duck

There is a difference between Nature and OOP. In Nature, objects are reproduced only from like objects, like the birds and the bees (the chicken and the egg are the exceptions to this rule.) However, in OOP instances are only created using the class template. In a pure OOP like Smalltalk, an instance of a specific class is created by sending the class a message. In fact, the

heart of OOP involves sending different types of messages from one object to another, even from an object to itself.

In order to see how messages work, let's start by entering the following commands to create a user-defined DUCK class and check that it's entered. Notice that no **role** descriptor is specified for the DUCK class. If a class has **role concrete**, direct instances of the class can be created. If the role is unspecified, CLIPS determines the role by inheritance. For determining role by inheritance, system classes behave as concrete classes. Thus by default, any class inheriting from USER is a concrete class and does not need to be declared as such in order to allow direct instances to be created.

If a class has **role abstract** no direct instances of it can be made. Abstract classes are defined for inheritance purposes only. For example, an abstract class called PERSON could be defined whose properties such as name, address, age, height, weight, and so on are inherited by concrete classes MAN and WOMAN. A direct instance of MAN could be a man-person called Harold, and a direct instance of WOMAN is a woman-person called Henrietta.

```
CLIPS> (defclass DUCK (is-a USER))
CLIPS> (describe-class DUCK)
===============================================================
==
***************************************************************
**
Concrete: direct instances of this class can be created.
Reactive: direct instances of this class can match defrule
patterns.

Direct Superclasses: USER
Inheritance Precedence: DUCK USER OBJECT
Direct Subclasses:
---------------------------------------------------------------
--
Recognized message-handlers:
init primary in class USER
delete primary in class USER
create primary in class USER
print primary in class USER
direct-modify primary in class USER
message-modify primary in class USER
```

```
direct-duplicate primary in class USER
message-duplicate primary in class USER
***********************************************************
**
============================================================
==
CLIPS>
```

Since classes are not objects in CLIPS, we can't send a message to make an object. Instead, the **make-instance** function is used to make an instance object. The basic syntax is as follows.

```
(make-instance [<instance-name>] of <class>
    <slot-override>)
```

Normally, you specify an instance-name. However, If you do not, CLIPS will generate one using the gensym* function. Slot values also can be specified.

Now that we have a duck factory, let's make some instances as follows, where the name of the instance is in brackets. Note the use of the "**of**" keyword to separate the instance name from the class name. You must include the "of" or a syntax error will result. Also, note that the brackets in the code mean an instance name, while brackets in the metasyntax such as for (make-instance) above, mean *optional*.

```
CLIPS> (make-instance [Dorky] of DUCK)
[Dorky]
CLIPS> (make-instance [Elsie] of COW)
[PRNTUTIL1] Unable to find class COW.
CLIPS> (make-instance Dorky_Duck of DUCK)
[Dorky_Duck]
CLIPS> (instances)
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 2 instances.
CLIPS>
```

After the instance is successfully created, CLIPS responds with the name of the instance. If it is *not* possible to create an instance, CLIPS responds with a FALSE. Also, like the (rules) and (facts) commands, CLIPS has an **instances** function to print out the instances of a class.

For the case of (make-instance), the brackets around the instance name is optional for a USER-defined class. As an example, let's create Dorky's cousin, Dorky_Duck, without brackets as follows.

```
CLIPS> (make-instance Dorky_Duck of DUCK)
; Instance Dorky_Duck is made.
[Dorky_Duck]
CLIPS>
```

There are two important rules about instances to keep in mind.

• Only one instance of the same name may be used in a module.

• A class cannot be redefined if instances of the class exist.

For example, let's make a clone of Dorky_Duck as follows. (It's this evil clone that always gets Dorky_Duck in trouble because no one can tell them apart. Kids often have clones like this too, and some adults.)

```
CLIPS> (make-instance Dorky_Duck of DUCK)
[Dorky_Duck]
CLIPS> (instances)
[Dorky] of DUCK
[Dorky_Duck] of DUCK
; Still only one Dorky_Duck
For a total of 2 instances.
CLIPS>
```

Much Ado About Instances

If a (reset) command is issued, all the instances in memory are deleted.

```
CLIPS> (reset)
CLIPS> (instances)
CLIPS>
```

Just as (deffacts) defines facts, there is also a **definstances** to define instances when a (reset) is issued. The following (definstances) also illustrates the optional comment in double quotes after the instance name, DORKY_OBJECTS.

```
CLIPS>
(definstances DORKY_OBJECTS "The Dorky Cousins"
```

```
        (Dorky of DUCK)
        (Dorky_Duck of DUCK))
CLIPS> (instances)
CLIPS> (reset)
CLIPS> (instances)
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 2 instances.
CLIPS>
```

The Disappearing Duck

Although a (reset) will delete all instances, it will also make new instances from (definstances). If you want to permanently delete an instance, the function **unmake-instance** will delete one or all instances, depending on its argument. To delete *all* instances, use the "*****".

The following examples illustrate the (unmake-instance) command.

```
; Delete all instances
CLIPS> (unmake-instance *)
TRUE
; Check that all are gone
CLIPS> (instances)
; Create new instances again
CLIPS> (reset)
; Check new instances created
CLIPS> (instances)
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 2 instances.
; Delete a specific instance
CLIPS> (unmake-instance [Dorky])
TRUE
CLIPS> (instances)
[Dorky_Duck] of DUCK
For a total of 1 instance.
CLIPS>
```

Another way to delete a specific instance is to **send** a **delete** message. The general syntax of the (send) function is as follows.

```
(send <instance-name> <message>)
```

Only one instance name can be specified in a command and it must be surrounded by brackets if it is a user-defined name.

For example, the following will make Dorky_Duck disappear.

```
; Create new instances again
CLIPS> (reset)
; Check new instances created
CLIPS> (instances)
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 2 instances.
CLIPS> (send [Dorky_Duck] delete)
TRUE
CLIPS> (instances)
[Dorky] of DUCK
For a total of 1 instance.
CLIPS>
```

The "***" in a (send) will *not* work to delete all instances. The "***" only works with the (unmake) function. Another alternative is to define your own handler for delete that will accept the "***" and thus allow you to (send <instance-name> my_delete *) messages.

A (send) message is acted upon only by a **target object** which has an appropriate handler. CLIPS automatically provides handlers for print, init, delete and so on for each user-defined class. It's important to realize that the message (send [Dorky_Duck] delete) works only because *this instance is a user-defined class*. If you define classes which do *not* inherit from USER such as a subclass of INTEGER, you *must* also create appropriate handlers to carry out all desired tasks such as printing, creating, and deleting instances. It's much easier to define subclasses of USER and take advantage of system-supplied handlers.

What Did You Have For Breakfast

The (send) function is the heart of OOP operation since it is the *only proper way for objects to communicate*. According to the principle of object encapsulation, one object should only be allowed to access another object's data by sending a message.

For example, if someone wants to know what you had for breakfast, they'll generally ask you, i.e., send a message. An impolite alternative would be to yank open your mouth and peer down your throat. If the principle of object encapsulation is not followed, any object can fool around with the private parts of other objects, with potentially disastrous results.

One useful application of (send) is to print information about an object. So far all the examples of objects that you have seen have no structure. However, just as deftemplate gives structure to a rule pattern, the slots give an object structure. For both deftemplate and objects, a slot is a named location in which data can be stored. However, unlike deftemplate slots, objects obtain their slots from classes, and classes use inheritance. Thus, the information in object slots can be effectively inherited by objects of subclasses. An **unbound** slot is one that has no values assigned. All slots *must* be bound.

As a simple example, let's make an object with slots to hold personal information and then send messages to it. The following commands will first set up the CLIPS environment with the appropriate constructs. The slots named *sound* and *age* initially contain no data, i.e., *nil* values.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
    (slot sound)
    (slot age))
CLIPS>
(definstances DORK_OBJECTS
    (Dorky_Duck of DUCK))
CLIPS> (reset)
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound nil)
(age nil)
CLIPS>
(send [Dorky_Duck] put-sound quack)
quack
CLIPS> (send [Dorky_Duck] print)
```

```
[Dorky_Duck] of DUCK
(sound quack)
(age nil)
CLIPS>
```

Notice that the slots are printed in the order defined in the class. However, if the instance inherits slots from more than one class, the slots from the more general classes will be printed first.

The value of a slot is changed using the put- message. By default, CLIPS creates a put-handler for each slot of a class to handle put- messages. Notice the dash, "-", at the end of "put-". The dash is an essential part of the message syntax since it separates the "put" from the slot name. Only one "put-" is allowed in a (send). Thus, to change multiple slots or the same slot of many instances, you must send multiple messages. Instead of doing this manually, it's possible to write a function to do multiple sends, or use the **modify-instance** function.

The value of a slot can be set by a slot-override in a make-instance. As an example,

```
CLIPS>
(make-instance Dixie_Duck of DUCK
    (sound quack) (age 2))
[Dixie_Duck]
CLIPS> (send [Dixie_Duck] print)
[Dixie_Duck] of DUCK
(sound quack)
(age 2)
CLIPS>
```

The complementary message to "put-" is *get-* which gets the data from a slot, as shown in the following example. If a *put-* is successful it returns the new value, while if *get-* is successful it returns the appropriate data. If the *put-* or *get-* does not succeed, an error message will be returned. The following examples show how this works.

```
; No slot color
CLIPS>
(send [Dorky_Duck] put-color white)
[MSGFUN1] No applicable primary message-handlers found for
put-color.
FALSE
CLIPS> (send [Dorky_Duck] get-age)
nil
; Value put in age
```

```
CLIPS> (send [Dorky_Duck] put-age 1)
1
; Check value is correct
CLIPS> (send [Dorky_Duck] get-age)
1
CLIPS>
```

In contrast to the *put-* message, the *get-* message *returns* the value of a slot. Since the value of *get-* is returned, it can be used by another function, assigned to a variable, and so forth. In contrast, a value that is printed out cannot be used by another function, assigned to a variable, and so forth because the value goes to the standard output device. One way of getting around this problem is to print out to a file and then read in the data from the file. While this is not an elegant solution, it does work. Another way is to write your own print message-handler that also returns values.

A very important point about slots is that you cannot modify the slots of a class by adding slots, deleting slots, or changing the characteristics of slots. The only way to change a class is to (1) delete all instances of the class, and (2) use a (defclass) with the same class name and the desired slots. This situation is analogous to modifying a rule by loading in a new rule with the same name.

Class Etiquette

Now that you've learned about slots and instances, it's time to discuss **class etiquette**. The term *etiquette* refers to a set of guidelines for doing something.

In contrast to standard procedural programming, the OOP paradigm is **class oriented**. Each object is intrinsically related to a class, and that class is part of a class hierarchy. Rather than concentrating on actions first and foremost, the OOP programmer considers the overall class hierarchy or **class architecture**, and how messages will be sent between objects. Thus, actions in customary procedural programs are performed explicitly, while the actions are performed implicitly in OOP. In either case, the end result is the same. However, the OOP system can be more easily verified, validated, and maintained.

The proper use of classes is summarized in the following *Rules of Class Etiquette*:

1. The class hierarchy should be in specialized logical increments using is-a links.

2. A class is unnecessary if it has only one instance.

3. A class should not be named for an instance and vice versa.

The first rule discourages the creation of a single class for your application. If a single class is adequate, then you probably don't need OOP. By creating classes in increments, you can more easily verify, validate, and maintain your code. In addition, incremental class hierarchies can be easily put in class libraries to greatly facilitate the creation of new code. This concept of class libraries is analogous to subroutine libraries for actions. Only is-a links can be used since this is the only relationship that Version 6.3 of CLIPS supports.

The second rule encourages the idea that classes are intended as a template to produce multiple objects of the same kind. Of course you can start out with zero or one instance. However, if you'll never need more than one instance in a class, you should consider modifying its superclass to accommodate the instance rather than defining a new subclass. If all your classes only have one instance, it is probable that your application is simply not well-suited to OOP and that coding in a procedural language may be best.

The third rule means that classes should not be named after instances and *vice versa*, to eliminate confusion.

Other Features

There are a number of useful predicate functions for slots. If you use these predicate functions to test for appropriate values to functions, your program will be more robust against crashes. In general, if a function does not return TRUE, it returns FALSE. Slot functions provided by CLIPS include:

class-slot-exists: Returns TRUE if the class slot exists.

slot-existp: Returns TRUE if the instance slot exists.

slot-boundp: Returns TRUE if the specified slot has a value.

instance-address: Returns the machine address at which the specified instance is stored.

instance-name: Returns the name given an instance.

instancep: Returns TRUE if its argument is an instance.

instance-addressp: Returns TRUE if its argument is an instance address.

instance-namep: Returns TRUE if its argument is an instance name.

instance-existp: Returns TRUE if the instance exists.

list-definstances: Lists all the definstances.

ppdefinstances: Pretty-prints the definstance.

watch instances: Allows you to watch instances being created and deleted.

unwatch instances: Turns off watching instances.

save-instances: Saves instances to a file.

load-instances: Loads instances from a file.

undefinstances: Deletes the named definstance.

Chapter 10

Fascinating Facets

If you want to have class, then act, dress, and talk like your friends.

In this chapter you'll learn more about slots and how to specify their characteristics by using **facets**. Just as slots describe instances, facets describe slots. The use of facets is good software engineering because there is a greater chance of CLIPS flagging an illegal value rather than risking a runtime error or crash. There are many types of facets that may be used to specify slots, as summarized in the following list:

default and **default-dynamic**: Set initial values for slots.

cardinality: Number of multifield values.

access: Read-write, read-only, initialize-only access.

storage: Local slot in instance or shared slot in class.

propagation: Inherit or no inherit slots.

source: Composite or exclusive inheritance.

override-message: Indicates message to send for slot override.

create-accessor: Create put- and get- handler.

visibility: Public or private to defining class only.

reactive: Changes to a slot trigger pattern-matching.

For reasons of space, we'll only describe a few facets in more detail in the rest of this chapter. For more details, see the *CLIPS Reference Manual*.

A Slot Named Default

The **default facet** sets the default value of a slot when an instance is created or initialized, as shown in the following example.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
```

```
      (slot sound (default quack))
      (slot ID)
      (slot sex (default male)))
CLIPS> (make-instance Dorky_Duck of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID nil)
(sex male)
CLIPS>
```

As you can see, the default values for slot sex and slot sound were set by the default facet values. Following the default keyword can be any valid CLIPS expression that does not involve a variable. For example, the default expression of the sound slot is the symbol quack. Functions may be used in the **facet expression** as will be shown in the next example.

This default facet is a **static default** because the value of its facet expression is determined when the class is defined and never changed unless the class is redefined. For example, let's set the default value of slot ID to the (gensym*) function which returns a new value not in the system every time it's called.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
     (slot sound (default quack))
     (slot ID (default (gensym*)))
     (slot sex (default male)))
CLIPS> ; Dorky_Duck #1
(make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID gen1)
(sex male)
CLIPS> ; Dorky_Duck #2
(make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
```

```
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID gen1)
(sex male)
CLIPS>
```

As you can see, the ID is always gen1 since (gensym*) is only evaluated once, and not again when the second instance is created. Note that the (gensym*) values may be different on your computer if you have already called (gensym*) since it increments by one each time it is called, and is *not* reset by a (clear). The (gensym*) function is reset to its starting value only if you restart CLIPS.

Now suppose that we want to keep track of the different Dorky_Duck instances that have been created. Rather than using the static default, we can use the facet called **default dynamic** which will evaluate its facet expression every time a new instance is created. Notice the difference between the following example and the previous.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
    (slot sound (default quack))
    (slot ID (default-dynamic (gensym*)))
    (slot sex (default male)))
CLIPS> ; Dorky_Duck #1
(make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID gen2)
(sex male)
CLIPS> ; Dorky_Duck #2
(make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID gen3)   ; Note ID is different
```

```
    (sex male) ; from Dorky_Duck #1
    CLIPS>
```

In this example which uses dynamic default, the ID of the second instance, gen3, is different from the first instance, gen2. In contrast, for the previous example of static default, the ID values were the same, gen1, since the (gensym*) was only evaluated once when the class was defined rather then for every new instance in the case of dynamic default.

Cardinal Properties

The **cardinality** of a slot refers to one of two types of fields that a slot can hold: (1) single-field, or (2) multifield. The term *cardinality* refers to a count. A bound single-field slot contains only one field, while a bound multifield slot may contain zero or more fields. The bound single-field slot and the bound multifield slot each contain *one* value. However, the one multifield value may have multiple fields in it. For example, (a b c) is a single multifield value with three fields. The empty string "" is a single-field value, just as "a b c" is. In contrast, an unbound slot has no value.

As an analogy to single and multifield variables, think of a slot as your mailbox. Sometimes you may find a single piece of junk-mail that doesn't have an envelope. Instead, an address label has just been stuck on a folded piece of paper addressed to "Resident." Other times you may find an envelope with multiple ads in it. The single piece of junk mail without an envelope is like a single-field value while the envelope with multiple ads is like the multifield value. If the junk-mail distributor slips up and mails you an envelope with nothing inside, this corresponds to the empty multifield variable. (Come to think of it, if the junk-mail envelope is empty, have you really received junk-mail?)

A **multiple facet** with keyword **multislot**, is used to store a multifield value as shown in the following example.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
    (multislot sound (default quack quack)))
CLIPS> (make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack quack)
CLIPS>
```

A multifield value can be accessed using get- and put- as shown in the following examples, which shows how to keep track of quacks.

```
CLIPS>
(send [Dorky_Duck] put-sound
                    quack1 quack2 quack3)
(quack1 quack2 quack3)
CLIPS> (send [Dorky_Duck] get-sound)
(quack1 quack2 quack3)
CLIPS>
```

Standard CLIPS functions such as nth$ to get the nth field of a multislot value can be used. The following example shows how to pick a certain quack.

```
CLIPS> (nth$ 1 (send [Dorky_Duck] get-sound))
quack1
CLIPS> (nth$ 2 (send [Dorky_Duck] get-sound))
quack2
CLIPS> (nth$ 3 (send [Dorky_Duck] get-sound))
quack3
CLIPS>
```

Other Features

CLIPS has several **multifield slot functions** as shown in the following list.

slot-replace$: Replace the specified range.

slot-insert$: Insert the specified range.

slot-delete$: Delete the specified range.

A multifield slot with no values, e.g., the empty multifield value (), may be assigned to a slot with a (multiple) facet. Note that there is a difference between a slot with an empty multifield value () and an unbound slot. If you think of an empty multifield value as analogous to an empty bus, you can see there is a difference between no people (unbound slot) and a bus with no people (empty multifield value, ()).

A **create-accessor** facet tells CLIPS whether to create **put-** and **get-** handlers for a slot. By default, all slots have a **read-write** create-accessor so you don't actually have to specify this facet to create handlers. If you define your own handlers, then you need to use ?NONE with the create-accessor facet. The other facet types for *create accessor* are **read** and **write**.

A **storage facet** defines one of two places that a slot value is stored: (1) in an instance, or (2) in the class. A slot value stored in the instance is called **local** because the value is only known to the instance. Thus, different instances may exist that have different local slot values. In contrast, a slot value stored in a class is called **shared** because it is the same for *all* instances of the class.

A local value is specified by the **local facet**, which is the default for a slot. A shared value is specified by the **shared facet** and *all* instances with this slot type will have their slot value automatically changed if *one* changes. An **access facet** defines one of three types of access for a slot, whether you use the handlers created by CLIPS or else define your own. The default type, **read-write**, allows you to both read and write the value of the slot. The other types are **read-only** and **initialize-only**.

Another way to set the instance values is with the **initialize-instance** function. An (initialize-instance) can be called at any time to reset the default values and retain values in non-(default) slots.

A (reset) can be thought of as a **cold-initialization** since all values in non-(default) slots are cleared, while a (initialize-instance) can be considered a **warm-initialization** since non-(default) values are retained. Of course, only (definstances) can be cold-initialized since non-(definstances) will simply be deleted. Also, slot-overrides may be used in (initialize-instance) as the last example shows.

Two predicate functions are designed for use with the access facets. Both these predicate functions return an error message if the specified slot or instance does not exist. The **slot-writablep** is a predicate function which returns TRUE if a slot is writable and FALSE if it is not. The **slot-initablep** predicate function returns TRUE if the slot is initializable and FALSE if it is not. The term *initializable* means that it is *not* read-only.

The **inherit facet**, which is the default, specifies that indirect instances of a class may inherit this slot from the class. As you recall, the indirect instances of a class are the instances of its subclasses, while the direct instances are those defined specifically for the class. The indirect instances of a class are direct instances of the subclasses in which they are defined. For example, [Dorky_Duck] is a direct instance of DUCK and an indirect instance of USER which is the superclass of DUCK. The **no-inherit facet** specifies that only a direct instance has the class slot.

It's important to realize that the (no-inherit) facet only prohibits inheritance from the class and not from its superclasses. This means that an instance may still inherit from superclasses of the (no-inherit) class.

The **composite facet**. facet states that facets which are not *explicitly* defined in the highest precedence class take their facets from the next higher precedence class. If the facet is not explicitly set in the next higher precedence class *and it is composite too*, CLIPS tries the next higher and so on until the facet is defined or there are no more classes. If the next higher class is not composite, CLIPS does not check further. The opposite to the composite facet is the **exclusive facet**, which is the default. For more information, see the *CLIPS Reference Manual*.

Chapter 11

Handling Handlers

There are two steps in learning how to use a shovel: (1) finding which part is the handle, and (2) what to do with the handle.

Handlers are essential in OOP because they support object encapsulation. The only proper way that objects can respond to messages is by having an appropriate handler to receive the message and take appropriate action. In this chapter you'll learn the how messages are interpreted by objects. You'll see how to modify existing message-handlers, and how to write your own.

Your Primitive Self

So far you've learned about the static structure of classes through the inheritance hierarchy and slots. However, the dynamic part of an object is determined by its message-handlers, or handlers for short, which receive messages and performs actions in response. The handlers are responsible for the dynamic properties of an object, which determine its behavior. You've already used one handler many times — the *print* in a (send).

Polymorphism; is one of the five generally accepted features of a true OOP language. For example, the same type of message, (send <instance-name> print), may have different actions depending on the class of the object which receives it. DUCK objects may print out sound, ID, and age, while DUCKLING objects only print out sound and age.

In languages without polymorphism, you would have to define one function, (send-duck-print) for duck types and another function, (send-duckling-print) for the duckling type. However, in OOP, no matter how many classes are defined, the same message, (send <instance-name> print), will print the object slots. This greatly improves the efficiency of program development since you do not have to define new functions for every new type.

Polymorphism can be carried to extremes by having the same message do completely different things. For example, a print message-handler could be defined that would *print* objects of a certain class and *delete* objects of another class. Another extreme possibility would be not having the print handler print anything. Instead, it would delete objects of one class, save objects of another class, add objects of another class, and so forth.

This extreme use of polymorphism would create a programmer's Tower of Babel and make it very hard to understand the code since everything would be run-time dependent. Defining message-handlers of the same name in different classes that do completely different things goes against the Principle of Least Astonishment.

Another example of polymorphism could be defined using a message-handler for "+". If a message for "+" is sent to strings or symbols, i.e., LEXEME objects, they will be concatenated because of a handler defined for the LEXEME class. If the "+" is sent to ordinary numbers, the result is addition because of the predefined system handler for addition of numbers. If the "+" is sent to complex numbers, defined as a subclass of USER called COMPLEX, a handler defined for the COMPLEX class will do complex number addition. Thus, the "+" does related types of operations that matches our intuition, and does not surprise us.

In addition to the predefined handlers such as *print*, you may define your own handlers. Let's start by writing a handler to add numbers through messages.

As you can see from Fig. 8.6 of chapter 8, the class NUMBER has subclasses INTEGER and FLOAT. Since these are predefined classes, it would seen natural to do numeric calculations by sending messages to numbers. Let's try it as follows.

```
CLIPS> (clear)
CLIPS> (send 1 + 2)
[MSGFUN1] No applicable primary message-handlers found for +.
FALSE
CLIPS>
```

Well, as you can see, this example didn't work. The reason why is implied in the error message. Let's check it out by obtaining more information about the INTEGER class since that was the target object of the print message.

```
CLIPS> (describe-class INTEGER)
================================================================
==
****************************************************************
**
Abstract: direct instances of this class cannot be created.

Direct Superclasses: NUMBER
Inheritance Precedence: INTEGER NUMBER PRIMITIVE OBJECT
Direct Subclasses:
****************************************************************
```

```
**
================================================================
==
CLIPS>
```

The problem is that the INTEGER class has no handler for "+". In fact, it has no handlers at all since none are listed. As you recall, the USER class and its subclasses always have print, delete, and other handlers automatically defined by CLIPS. If we want to send print messages to an object like 1 of the INTEGER class, we'll have to make our own print handler.

Before writing this handler, let's answer a question that you may have concerning how INTEGER can have instances. Since INTEGER is an abstract class, you may wonder how it can have instances such as 1, 2, 3, etc. Although you cannot make direct instances of an abstract class, you can make use of existing instances. For the case of the predefined system class INTEGER, all the integers up to the maximum allowed are available as objects. Likewise, strings and symbols are available for the system defined abstract classes STRING and SYMBOL, and so forth for the other predefined classes.

Shown following is the definition of a handler for the NUMBER class that will handle addition by messages. The handler is defined for NUMBER rather than INTEGER because we would also like to handle FLOAT objects too. Instead of defining the same handler for FLOAT and for INTEGER, it's easier to just define a handler for the superclass NUMBER. *If a handler for a message is not defined in the class of the object, CLIPS tries all the handlers in the inheritance precedence list.* Since "+" is not defined for INTEGER, CLIPS tries NUMBER next, finds the applicable handler, and returns the result of 3.

```
CLIPS>
; ?arg is argument of handler
(defmessage-handler NUMBER + (?arg)
   ; Function addition of handler
   (+ ?self ?arg))
CLIPS> (send 1 + 2)
3
CLIPS> (send 2.5 + 3)
5.5
CLIPS> (send 2.5 + 2.6)
5.1
CLIPS> (describe-class NUMBER)
================================================================
==
```

```
        **********************************************************
        **
        Abstract: direct instances of this class cannot be created.

        Direct Superclasses: PRIMITIVE
        Inheritance Precedence: NUMBER PRIMITIVE OBJECT
        Direct Subclasses: INTEGER FLOAT
        ----------------------------------------------------------
        --
        Recognized message-handlers:
        + primary in class NUMBER
        **********************************************************
        **

        ==========================================================
        ==
        CLIPS>
```

The variable **?self** is a special variable in which CLIPS stores the **active instance**. The *?self* is a reserved word that cannot be explicitly included in a handler argument, nor can it be bound to a different value. The active instance is the instance to which the message was sent. In our example, all the predefined classes such as NUMBER, INTEGER, and FLOAT are all subclasses of the PRIMITIVE class. This is in contrast to USER, which is the other main subclass of OBJECT.

As another example, lets write a handler to concatenate strings, symbols or both.

```
CLIPS>
; ?arg is argument of handler
(defmessage-handler LEXEME + (?arg)
   ; Function concatenation of handler
   (sym-cat ?self ?arg))
; SYMBOL + STRING
CLIPS> (send Dorky_ + "Duck")
Dorky_Duck ; SYMBOL result
CLIPS>
```

Notice that the handler is defined for the LEXEME class since that is a superclass of both SYMBOL and STRING. In this case, the handler returns a SYMBOL since (sym-cat) is used.

This example also illustrates why brackets may be necessary in a (send). As shown in this example, the message goes to the SYMBOL *Dorky_* without brackets. With brackets, the message goes to an object *[Dorky_]* of a user-defined class. Here we assume that *[Dorky_]* could be an object of a user-defined class, such as DUCK.

Make 'Em Pay Through the Nose

The real utility of handlers is with subclasses of USER since you can define instances of these classes. To see how handlers work in this case, let's first set up the environment as follows.

```
CLIPS> (clear)
CLIPS>
(defclass DUCKLING (is-a USER)
    (slot sound (default quack))
    (slot age (visibility public)))
CLIPS>
(defclass DUCK (is-a DUCKLING)
    (slot sound (default quack)))
CLIPS>
(definstances DUCKY_OBJECTS
    (Dorky_Duck of DUCK (age 2))
    (Dinky_Duck of DUCKLING (age .1)))
CLIPS> (reset)
CLIPS>
```

As a simple example, let's write a handler that will print out the slots of the active instance. We can make use of the **ppinstance** function to print out the slots of the active instance. This function does *not* return a value and is used only for its side-effect of printing to the standard device. Also, it can only be used from within a handler since only there is the active instance known. Shown following is a USER-defined handler called print-slots that prints out the slots of the active instance using (ppinstance).

```
CLIPS>
(defmessage-handler USER print-slots ()
    (ppinstance))
CLIPS> (send [Dorky_Duck] print-slots)
[Dorky_Duck] of DUCK
(age 2)
```

```
   (sound quack)
   CLIPS>
```

Although the handler could be defined just for DUCK in this case, a handler defined for USER will be called for all subclasses of USER, not just DUCKLING and DUCK. Thus, the handler print-slots will work for *all* subclasses that we may define of USER.

Of course it's possible to get carried away and define all your message handlers as USER handlers. However, it's good style and improves efficiency to define handlers as close as possible to the class or classes for which they are intended. Efficiency is improved because CLIPS does not have to keep searching through a lot of classes to find the applicable handler.

Getting Around

Let's examine message-handlers in more detail now. We'll define a handler to print out a header when an object receives a message to print itself. The message-handler is defined using a defmessage-handler construct as follows.

```
   CLIPS>
   (defmessage-handler USER print before ()
       (printout t "*** Starting to print ***"
                 crlf))
   CLIPS> (send [Dorky_Duck] print)
   *** Starting to print ***
   [Dorky_Duck] of DUCK
   (age 2)
   (sound quack)
   CLIPS>
```

The reason that a header is printed rather than a trailer at the end has to do with the **handler type**. A **before** type handler is used *before* the print message. To make a trailer, use the **after** type handler as shown in the following example.

```
   CLIPS>
   (defmessage-handler USER print after ()
       (printout t "*** Finished printing ***"
                 crlf))
   CLIPS> (send [Dorky_Duck] print)
   *** Starting to print ***
   [Dorky_Duck] of DUCK
```

```
(age 2)
(sound quack)
*** Finished printing ***
CLIPS>
```

The general format of a message-handler is as follows.

```
(defmessage-handler <class-name>
   <message-name> [handler-type]
   [comment]
   (<parameters>* [wildcard-parameter])
   <action>*)
```

While there may be multiple actions in a handler, *only the value of the last action is returned*. Notice that this is just like a (deffunction).

Since [Dorky_Duck] is of class DUCK, a subclass of USER, we can take advantage of the **print** handler that is predefined by CLIPS for the USER class. All subclasses of USER can take advantage of the handlers of USER, which saves you the trouble of writing handlers for every class that you define. Notice how the concept of inheritance from USER to its subclass DUCK simplifies program development by allowing reuse of existing code, i.e., the print handler of USER.

The empty parentheses () that follow the *before* handler type mean that there are neither parameters nor a wildcard parameter. In other words, the header handler takes no arguments and so the parentheses are empty, but required. Note that while multiple parameters may be used, there can be only one wildcard.

Primary Considerations

As you can see, the trailer handler is the same as the header handler except that an *after* handler type is used, and the action text is different. Thus, a before handler type does its task before the **primary** type handler, and an after handler does its task after the primary handler. A *primary* is intended to do the major task. An around handler type is intended to set up the environment for the rest of the handlers. The before and after types are intended for minor tasks such as initializing variables or printing, while the primary does the major task.

The following list summarizes the class role and return value of each handler type:

around: Set up environment for other handlers. Returns a value.

before: Auxillary work before primary. No return value.

primary: Perform major task of message. Returns a value.

after: Auxillary work after primary. No return value.

The handler types are listed in the order that they are normally called during execution of a message. Depending on the handler type, CLIPS knows when to execute it. That is, an **around** handler starts before any before handlers. A before handler is executed before any primary handlers, which are followed by the after handlers. The exception to this sequential handler execution is the around type handler. If an around handler is defined, it will start execution before any of the others, perform specified actions, and then complete its actions *after* all the other handler types have finished. You'll see a detailed example of these handlers execution soon.

The **class role** describes the intended purpose of each type. The return value describes whether the handler type is generally intended for a **return value** or simply to provide a useful side-effect such as printing**.** This consideration will depend on the handler. For example, many user-defined primary handlers may be written to return a value as the result of some numeric calculation or string operation. An exception to returning a useful return value is a print primary handler whose main task is the side-effect of printing, and does not have a return value.

The list of predefined primary handlers of USER and their class role is shown following. By inheritance, these are available for all subclasses of USER.

init: Initializes an instance.

delete: Deletes an instance.

print: Prints an instance.

direct-modify: Directly modifies slots.

message-modify: Modifies slots using put- messages.

direct-duplicate: Duplicates an instance without using put- messages.

message-duplicate: Duplicates an instance using messages.

These primary handlers are predefined and cannot be modified unless you change the source code of CLIPS. However, you may define the *before*, *after* and *around* handler types for these primaries. You've already seen an example of changing the before and after handler types for the USER print handler. Now let's look at some examples of defining the before and after handler types for the **init** primary handler**.**

```
CLIPS>
(defmessage-handler USER init before ()
   (printout t
      "*** Starting to make instance ***"
```

```
        crlf))
CLIPS>
(defmessage-handler USER init after ()
    (printout t
        "*** Finished making instance ***"
        crlf))
CLIPS> (reset)
*** Starting to make instance ***
*** Finished making instance ***
*** Starting to make instance ***
*** Finished making instance ***
CLIPS>
(make-instance Dixie_Duck of DUCK
    (age 1))
*** Starting to make instance ***
*** Finished making instance ***
[Dixie_Duck]
CLIPS> (instances)
[Dorky_Duck] of DUCK
[Dinky_Duck] of DUCKLING
[Dixie_Duck] of DUCK
For a total of 3 instances.
CLIPS>
```

The Power of Belief

The self parameter is useful because it can be used to *read* a slot value in the form

```
?self:<slot-name>
```

It can also be used to *write* a slot value using the bind function. The "?self:" notation is more efficient than sending messages but can only be used from within a handler on its active instance. In contrast, the **dynamic-get-** and **dynamic-put-** functions can be used from within a handler to read and write a slot value of the active instance. Although you can use messages from inside a handler, such as the following, it's not efficient.

```
(send ?self dynamic-get-<slot>)
(send ?self dynamic-put-<slot>)
```

As an example of dynamic-get-, let's change our example as follows.

```
CLIPS>
(defmessage-handler USER print-age primary ()
    (printout t "*** Starting to print ***"
              crlf
              "Age = " (dynamic-get age)
              crlf
              "*** Finished printing ***"
              crlf))
CLIPS> (send [Dorky_Duck] print-age)
*** Starting to print ***
Age = 2
*** Finished printing ***
CLIPS>
```

The ?self:age can only be used in a class and its subclasses which inherit the slot age. The ?self:<slot-name> is evaluated in a *static* way through inheritance. This means that if a subclass redefines a slot, a superclass message-handler will fail if it tries to directly access the slot using ?self:<slot-name>.

In contrast, the dynamic-get- and dynamic-put- can be used by superclasses and subclasses because these check slots *dynamically*. In order for a superclass to dynamically reference a slot, however, the visibility facet of the slot must be public. The following example would *not* work if DUCK is changed to USER.

```
CLIPS>
(defmessage-handler DUCK print-age primary ()
    (printout t "*** Starting to print ***"
              crlf
              "Age = " ?self:age
              crlf
              "*** Finished printing ***"
              crlf))
CLIPS> (send [Dorky_Duck] print-age)
*** Starting to print ***
Age = 2
*** Finished printing ***
CLIPS>
```

As an example of using a dynamic-put- function in a handler, suppose we want to help Dorky_Duck regain some of his youth. The following example shows how his age can be

changed using a handler. This example also illustrates how a value can be passed to a handler through the ?change variable.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
    (slot sound (default quack))
    (slot age))
CLIPS>
(defmessage-handler DUCK lie-about-age
    (?change)
    (bind ?new-age (- ?self:age ?change))
    (dynamic-put age ?new-age)
    (printout t "*** Starting to print ***"
                crlf
                "I am only " ?new-age
                crlf
                "*** Finished printing ***"
                crlf))
CLIPS>
(make-instance [Dorky_Duck] of DUCK (age 3))
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] lie-about-age 2)
*** Starting to print ***
I am only 1
*** Finished printing ***
CLIPS>
```

As you can see, Dorky_Duck's belief is so strong that the changed age is put in his age slot. Notice how the handler uses the variable ?new-age to store the changed age, which is then put into the age slot of the instance.

The Truthful Daemon

A **daemon** is a handler which executes whenever some **basic action** like initialization, deletion, get, or put is performed on an instance. A rule cannot be considered a daemon because it's not certain that it will be executed just because its LHS patterns are satisfied. The only thing that is certain is that a rule will become *activated* when its LHS is satisfied — not that it will execute.

There is no explicit keyword for daemon since it's just a concept. The before and after handlers that printed strings can be considered print daemons. These handlers waited for a (send [Dorky_Duck] print-age) message to trigger their action. First the before handler printed its string, then the primary handler printed, and finally the after handler printed. One daemon is the before handler which waits for a print message. The second daemon is the after handler that waits for the print primary to finish printing.

Printing is not considered a basic action because there is no return value associated with a (send <instance> print). The print message is only sent for the side-effect of printing. In contrast, a (send <instance> get-<slot>) message will return a value that may be used by other code. Likewise, the initialization, deletion, and put all have an effect on an instance and so are considered basic actions like get.

Daemons are easily implemented using before and after handlers since these will be executed before and after their primary handler. Implementing daemons like this is called **declarative implementation** because no explicit actions on the part of the handler is necessary for it to be executed. That is, CLIPS will always execute a before handler before its primary and will always execute an after handler after its primary. In a declarative daemon implementation, the normal operation of CLIPS will cause the daemons to be activated when their time has come. Thus, the declarative implementation is implicit in the normal operation.

The opposite of implicit execution is the **imperative implementation** in which the actions are explicitly programmed. The around handler is very convenient to use for imperative daemons. The basic idea of the around handler is as follows.

1. Start before any other handlers.

2. Call the next handler using either **call-next-handler** to pass the same arguments or **override-next-handler** to pass different ones.

3. Continue execution when the last handler finishes.

4. After any other around, before, primary, or after handlers finish, the around handler resumes execution.

The keyword *call-next-handler* is used to call the next handler(s). A handler is said to be *shadowed* by a **shadower** if it must be called from the shadower by a function such as call-next-handler. The call-next-handler may be used multiple times to call the same handler(s).

A predicate function called **next-handlerp** is used to test for the existence of a handler before the call is made. If no handler exists, then next-handlerp will return FALSE.

The following example illustrates the around handler through a truthful daemon that tells on Dorky_Duck whenever he lies about his age.

```
CLIPS>
(defmessage-handler DUCK lie-about-age around
    (?change)
    (bind ?old-age ?self:age)
    (if (next-handlerp) then
        (call-next-handler))
    (bind ?new-age ?self:age)
    (if (<> ?old-age ?new-age) then
        (printout t "Dorky_Duck is lying!"
                    crlf
                    "Dorky_Duck is lying!"
                    crlf
                    "He's really " ?old-age
                    crlf)))
CLIPS>
(make-instance [Dorky_Duck] of DUCK (age 3))
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] lie-about-age 1)
*** Starting to print ***
I am only 2
*** Finished printing ***
Dorky_Duck is lying!
Dorky_Duck is lying!
He's really 3
CLIPS>
```

Although Dorky_Duck may still lie about his age, the daemon tells the truth.

Notice the ?change argument. Although the around handler does not use ?change, the lie-about-age primary that is called by the call-next-handler does need it to change the age. Thus, ?change must be passed to the primary by the around handler. An error message will occur if you leave out the ?change.

The call-next-handler *always* passes the arguments of the shadower to the shadowed handler. It's possible to pass *different* arguments to a shadowed handler by use of the override-next-handler function as shown in the following example.

```
CLIPS>
(defmessage-handler DUCK lie-about-age around
    (?change)
    (bind ?old-age ?self:age)
    (if (next-handlerp) then
        ; Divide age in half!
        (override-next-handler (/ ?change 2)))
    (bind ?new-age ?self:age)
    (if (<> ?old-age ?new-age) then
        (printout t "Dorky_Duck is lying!"
                    crlf
                    "Dorky_Duck is lying!"
                    crlf
                    "He's really " ?old-age
                    crlf)))
CLIPS>
(make-instance [Dorky_Duck] of DUCK (age 3))
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] lie-about-age 1)
*** Starting to print ***
I am only 2.5
*** Finished printing ***
Dorky_Duck is lying!
Dorky_Duck is lying!
He's really 3
CLIPS>
```

It's important to keep in mind that the return value of call-next-handler and override-next-handler is that of the shadowed handlers.

Shown following are the *Rules of Message Dispatch:*

1. All the around handlers start execution. Then the before, primary, and after handlers start and finish, followed by completion of the around's execution.

2. The around, before, and primary handlers are called in order of highest precedence class to lowest.

3. The after handlers are called from lower precedence class to highest.

4. Each around handler must explicitly call the next shadowed handler.

5. Higher precedence primaries must explicitly call lower precedence (shadowed) primaries if they are to execute.

However, note the following prerequisite to any message handling:

There must be at least one applicable primary handler.

Since only around and primary handlers can return values, and the around shadows primaries, it follows that the return value of a (send) will be the around return value. If there is no around, then the return value will be that of the highest precedence primary.

The following list summarizes the return value of the handler types.

around: Ignore or capture return value of next most specific around or primary.

before: Ignore. Side-effect only.

primary: Ignore or capture return value of most general primary scope.

after: Ignore. Side-effect only.

Get the Point

Up to now, we've discussed inheritance using only is-a links. As you've seen, this type of inheritance relationship is good for defining classes that are more and more specialized. That is, you start off by defining the most general classes as a subclass of USER, and then define more specialized classes with more slots in the lower levels of the class hierarchy.

Normally, you design new classes as specializations of existing ones. This paradigm is **Inheritance by Specialization**. As you recall from the quadrilateral example of Fig. 8.7 of chapter 8, the highest level is quadrilateral, and the lower levels are trapezoid, parallelogram, rectangle, and then square along one inheritance path. Trapezoid is a special class of quadrilateral, parallelogram is a special class of trapezoid, rectangle is a special class of parallelogram, and square is a special class of rectangle.

Inheritance can also be used to build up more complex classes. However, this is not quite as direct in CLIPS. As an example, the basic class for geometry is a POINT containing a single slot point1. A LINE can be then defined by adding point2 to POINT. A TRIANGLE is defined by adding point3 to LINE, and so on for QUADRILATERALS, PENTAGONS, etc.

```
(defclass POINT (is-a USER)
    (multislot point1))


(defclass LINE (is-a POINT)
```

```
      (multislot point2))

   (defclass TRIANGLE (is-a LINE)
      (multislot point3))
```

Notice how each class is a specialization of its parent class by inheriting the superclass points and then adding one new point.

The opposite paradigm is **Inheritance by Generalization** in which more general classes are built up from simple ones. For example, a LINE is considered made up of two points. A TRIANGLE is made up of three lines. A QUADRILATERAL is made up of four lines, and so forth. This would be a good paradigm for building an object-oriented drawing system in which new objects could be built up out of simpler ones.

There is a subtle but important difference between this example of specialization, and generalization. In specialization, new classes are built up by adding specialized slots which are the same type as superclass slots. Thus, POINT, LINE, and TRIANGLE all have point-type slots.

In contrast, generalization builds up using new types of slots defined for each class. Class POINT has a point-type slot. LINE has two point-type slots. TRIANGLE has three line-type slots. QUADRILATERAL has four line-type slots, and so on. Generalization is good for **synthesis**, which means a building up. The opposite of synthesis is **analysis**, which means taking apart or a simplification. The model for analysis is specialization.

Fig. 11.1 illustrates one inheritance scheme for polygons in which classes are built by inheritance. In this case, the link between classes would be "is-made-of". Thus, a LINE is-made-of POINT. A POLYGON is-made-of LINE, and so forth.

Fig. 11.1 Polygon Hierarchy Showing a Few Regular Polygon Classes
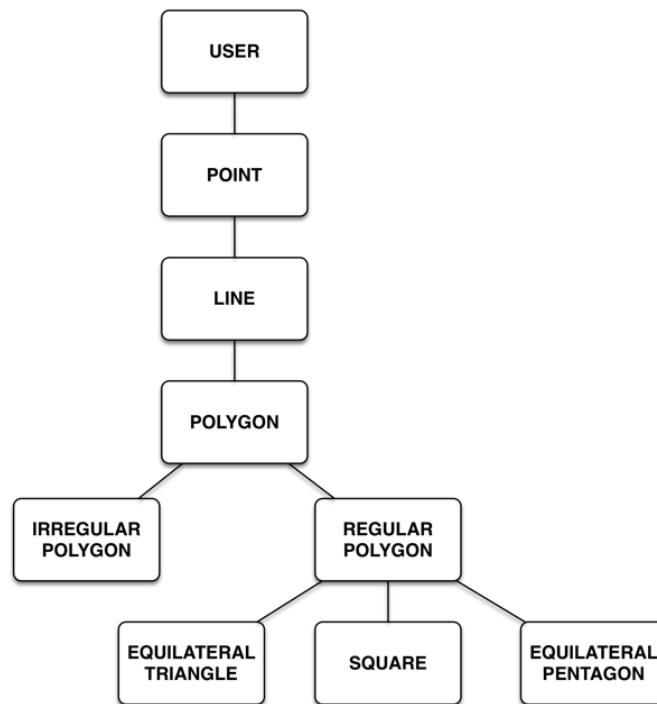
Links such as "is-made-of" can be simulated in CLIPS by appropriate slot definitions even though only is-a links are supported in Version 6.3. As an example of generalization, let's build up a LINE class as a generalization of a POINT class. The POINT class will provide instances that have a position. In order to make this example realistic, we'll assume an arbitrary number of dimensions by defining the position as (multiple). Thus, a one-dimensional point will have one value in the position slot, a two-dimensional point will have two values, and so forth. The definition of the POINT class is very simple.

```
CLIPS> (clear)
CLIPS>
(defclass POINT (is-a USER)
    (multislot position
        (propagation no-inherit)))
CLIPS>
```

The (no-inherit) facet is used to prevent a LINE from inheriting a position slot. Instead, a LINE will be defined by two points called slot point1 and slot point2. These two slots will define the line and it is extraneous to have an additional position slot by inheritance.

The definition of the LINE class is a little more complex. The reason for the added complexity is that the details of implementation are included in the (defclass) because Version 6.3 only supports is-a relationships.

```
(defclass LINE (is-a POINT)
    (slot point1
        (default-dynamic
            (make-instance (gensym*) of POINT))
        (propagation no-inherit))
    (slot point2
        (default-dynamic
            (make-instance (gensym*) of POINT))
        (propagation no-inherit))
    (message-handler find-point)
    (message-handler print-points)
    (message-handler find-distance)
    (message-handler print-distance))
```

Note that the message-handlers of LINE are **forward-declared** for documentation purposes.

At this time you may be wondering why POINT and LINE are not both defined as subclasses of USER since all their slots have (no-inherit) facets. Since all the slots of POINT, LINE, and the TRIANGLE class to be defined later have (no-inherit) facets, all these classes could be defined as direct subclasses of USER rather than defining LINE as a subclass of POINT and TRIANGLE as a subclass of LINE.

However, the whole point of this example is to illustrate Inheritance by Generalization, which is a logical concept that is not directly supported by Version 6.3. Thus, defining LINE as a subclass of POINT and TRIANGLE as a subclass of LINE is done for reasons of documenting the logical concept of Inheritance by Generalization. Admittedly, a comment could be added by the (defclass LINE (is-a USER)) and (defclass TRIANGLE (is-a USER)) stating that we are trying to implement Inheritance by Generalization, but seeing the code in place is better documentation. If Inheritance by Generalization is ever directly supported by CLIPS, these (defclass) statements will make it easy to convert.

The reason for including the (make-instance (gensym*)) in the LINE slots is to provide the inheritance from the POINT class. With the standard Inheritance by Specialization, only one position slot of LINE is possible because POINT has only one position slot. It is not possible for both slot point1 and slot point2 of LINE to inherit the position slot of POINT. The actual slot value of each LINE will be a gensym* value. Each gensym* value will be the instance name of a

point instance. The point position can then be accessed through the gensym* value. Thus, the gensym* values act as pointers to different instances.

This **indirect access** technique of (gensym*) values is analogous to using a pointer to access a value in a procedural language. Thus, the different slots of LINE can indirectly inherit the same slots of POINT. It's convenient to use (gensym*) because we don't care what the pointer names of LINE are, any more than we care what the pointer addresses are in a procedural language.

The following examples show how the points are accessed for one-dimensional points at position 0 and 1.

```
CLIPS>
(definstances LINE_OBJECTS
    (Line1 of LINE))
CLIPS> (reset)
CLIPS>
(send (send [Line1] get-point1)
     put-position 0)
(0)
CLIPS>
(send (send [Line1] get-point2)
     put-position 1)
(1)
CLIPS> (send [Line1] print)
[Line1] of LINE
(point1 [gen1])
(point2 [gen2])
CLIPS>
(send (send [Line1] get-point1)
     get-position)
(0)
CLIPS>
(send (send [Line1] get-point2)
     get-position)
(1)
CLIPS>
```

Now that you understand how the indirect access works, let's define some handlers to avoid the trouble of entering the nested (send) messages, as in the last two cases. Let's define a handler

called find-point to print out a specified point value, and a handler called print-points to print out the values of both LINE points as follows. The argument of find-point will be either a 1 for point1 or a 2 for point2.

```
CLIPS>
(defmessage-handler LINE find-point (?point)
    (send (send ?self
                (sym-cat "get-point" ?point))
          get-position))
CLIPS>
(defmessage-handler LINE print-points ()
    (printout t "point1 "
                (send ?self find-point 1)
                crlf
                "point2 "
                (send ?self find-point 2)
                crlf))
CLIPS> (send [Line1] find-point 1)
(0)
CLIPS> (send [Line1] find-point 2)
(1)
CLIPS>
```

For real use, it would be best to provide error detection so that only a 1 or 2 is allowed.

As you can see, the handler works fine for one-dimensional points. It can be tested for two-dimensional points as follows. We'll assume the first number for each point is the X-value, and the second number is the Y-value. That is, point1 has X-value 1 and Y-value 2.

```
CLIPS>
(send (send [Line1] get-point1)
      put-position 1 2)
(1 2)
CLIPS>
(send (send [Line1] get-point2)
      put-position 4 6)
(4 6)
CLIPS> (send [Line1] print)
[Line1] of LINE
(point1 [gen1])
```

```
    (point2 [gen2])
CLIPS>
(send (send [Line1] get-point1)
      get-position)
(1 2)
CLIPS>
(send (send [Line1] get-point2)
      get-position)
(4 6)
CLIPS>
```

As expected, the handler also works correctly for two-dimensional points.

Now that we have the two point positions, it's easy to find the distance between the points, which is the length of the line. The distance can be determined by defining a new handler called find-distance which uses the Pythagorean Theorem to calculate the distance as the square root of the sum of the squares. Since no assumptions were made as to the number of dimensions, the (nth$) function is used to pick out each multifield value up to the maximum number of coordinates, as stored in the ?len variable.

```
CLIPS>
(defmessage-handler LINE find-distance ()
    (bind ?sum 0)
    (bind ?index 1)
    (bind ?len
        (length$ (send ?self find-point 1)))
    (bind ?Point1 (send ?self find-point 1))
    (bind ?Point2 (send ?self find-point 2))
    (while (<= ?index ?len)
        (bind ?dif (- (nth$ ?index ?Point1)
                      (nth$ ?index ?Point2)))
        (bind ?sum (+ ?sum (* ?dif ?dif)))
        (bind ?index (+ ?index 1)))
    (bind ?distance (sqrt ?sum)))
CLIPS>
(defmessage-handler LINE print-distance ()
    (printout t "Distance = "
              (send ?self find-distance)
              crlf))
```

```
CLIPS> (send [Line1] print-distance)
Distance = 5.0
CLIPS>
```

The values 1, 2 for point1 and 4, 6 for point2 were chosen for an easy check of the handler since these coordinates define a 3–4–5 triangle. As you can see, the distance is 5.0, as expected.

Treasure Maps

Now that the POINT and LINE classes have been defined by generalized inheritance, why stop now? Let's continue with the next simplest class that can be defined from a line — the triangle. Shown following are the three defclasses required.

```
CLIPS> (clear)
CLIPS>
(defclass POINT (is-a USER)
    (multislot position
        (propagation no-inherit)))
CLIPS>
(defclass LINE (is-a POINT)
    (slot point1
        (default-dynamic
            (make-instance (gensym*) of POINT))
        (propagation no-inherit))
    (slot point2
        (default-dynamic
            (make-instance (gensym*) of POINT))
        (propagation no-inherit)))
CLIPS>
(defclass TRIANGLE (is-a LINE)
    (slot line1
        (default-dynamic
            (make-instance (gensym*) of LINE))
        (propagation no-inherit))
    (slot line2
        (default-dynamic
            (make-instance (gensym*) of LINE))
        (propagation no-inherit))
    (slot line3
```

```
            (default-dynamic
                (make-instance (gensym*) of LINE))
            (propagation no-inherit)))
   CLIPS>
```

Notice that the (no-inherit) facets in TRIANGLE are technically not necessary since there is no subclass of TRIANGLE defined. The reason for including the (no-inherit) facets is because of **defensive-programming**, which is analogous to defensive-driving. If another subclass is added, either by you or someone else, the (defclass) of TRIANGLE must be modified to include the (no-inherit) facets. This is bad style because it means that existing, debugged code must be modified. If you're going to enhance existing, debugged code, you shouldn't have to modify it. It's better to plan ahead for enhancements.

Next we'll define a triangle instance and check the instances generated. Note that your gen values may be different from those shown unless you've just started or restarted CLIPS, or have not used (gensym*) or (gensym) since you started.

```
   CLIPS>
   (definstances TRIANGLE_OBJECTS
      (Triangle1 of TRIANGLE))
   CLIPS> (reset)
   CLIPS> (instances)
   [Triangle1] of TRIANGLE
   [gen3] of LINE
   [gen4] of POINT
   [gen5] of POINT
   [gen6] of LINE
   [gen7] of POINT
   [gen8] of POINT
   [gen9] of LINE
   [gen10] of POINT
   [gen11] of POINT
   For a total of 10 instances.
   CLIPS>
```

At first you may be surprised at all the gensym* values created. However, all of them are necessary. First, gen3 was created for slot line1, which required gen4 and gen5 for the slots point1 and point2 associated with it by inheritance. Second, gen6 was created for slot line2, which required gen7 and gen8 for its slots point1 and point2. Finally, gen9 was created for slot line3,

which required gen10 and gen11 for the slots point1 and point2 associated with it . The slots for [Triangle1] and one of its pointer values, [gen3], is shown following.

```
CLIPS> (send [Triangle1] print)
[Triangle1] of TRIANGLE
(line1 [gen3])
(line2 [gen6])
(line3 [gen9])
CLIPS> (send [gen3] print)
[gen3] of LINE
(point1 [gen4])
(point2 [gen5])
CLIPS>
```

Now lets put in the X, Y coordinates of [Triangle1] as follows.

```
CLIPS>
(send (send (send [Triangle1] get-line1)
            get-point1)
      put-position -1 0)
(-1 0)
CLIPS>
(send (send (send [Triangle1] get-line1)
            get-point2)
       put-position 0 2)
(0 2)
CLIPS>
(send (send (send [Triangle1] get-line2)
            get-point1)
       put-position 0 2)
(0 2)
CLIPS>
(send (send (send [Triangle1] get-line2)
            get-point2)
      put-position 1 0)
(1 0)
CLIPS>
(send (send (send [Triangle1] get-line3)
            get-point1)
```

```
        put-position 1 0)
(1 0)
CLIPS>
(send (send (send [Triangle1] get-line3)
            get-point2)
      put-position -1 0)
(-1 0)
CLIPS>
```

The stored values are as follows.

```
CLIPS> (send [Triangle1] print)
[Triangle1] of TRIANGLE
(line1 [gen3])
(line2 [gen6])
(line3 [gen9])
CLIPS> (send [gen3] print)
[gen3] of LINE
(point1 [gen4])
(point2 [gen5])
CLIPS> (send [gen4] print)
[gen4] of POINT
(position -1 0)
CLIPS> (send [gen5] print)
[gen5] of POINT
(position 0 2)
CLIPS>
```

As you can see, the line1 pointer [gen1] points to point1 and point2 with their pointers [gen2] and [gen3]. These last two pointers finally point to the actual values of (-1 0) and (0 2) that define line1 of [Triangle1]. It's analogous to Long John Silver finding a treasure chest with a map, [gen1], which leads to another chest with two maps, [gen2] and [gen3], which lead to the two buried treasures at the locations specified by [gen2] and [gen3].

The values stored for each line of [Triangle1] can be retrieved by a single command using nested messages such as the following.

```
(send (send (send [Triangle1] get-line1)
            get-point1)
      get-position)
```

Although these commands work, it's not much fun to type them in unless you get paid by the hour and need the typing practice. As you might have guessed from the LINE handlers that we defined in the previous section, it's possible to define TRIANGLE handlers as follows.

```
CLIPS>
(defmessage-handler TRIANGLE find-line-point
    (?line ?point)
    (send (send (send ?self
                      (sym-cat "get-line"
                               ?line))
                (sym-cat "get-point" ?point))
          get-position))
CLIPS>
(defmessage-handler TRIANGLE print-line
    (?line)
    (printout t "point1 "
                (send ?self find-line-point
                      ?line 1)
                crlf
                "point2 "
                (send ?self find-line-point
                      ?line 2)
                crlf))
CLIPS> (send [Triangle1] print-line 1)
point1 (-1 0)
point2 (0 2)
CLIPS>
```

Using these handlers is a lot more convenient than typing in the nested messages.

At this point, you might be tempted to define a handler called find-line which returns both point values of the specified line. Recall that find-line-point requires the specification of both the line and one of the two points which define the line. So why not just send two messages in the same handler to return both point values of the specified line? Shown following is the handler for find-line and what it returns for line1 of [Triangle1].

```
CLIPS>
(defmessage-handler TRIANGLE find-line (?line)
    (send (send (send ?self
                      (sym-cat "get-line"
```

```
                                    ?line))
                       get-point1)
              get-position)
        (send (send (send ?self
                             (sym-cat "get-line"
                                      ?line))
                       get-point2)
              get-position))
CLIPS> (send [Triangle1] find-line 1)
(0 2)
CLIPS>
```

As you can see, the handler only returns the last message value of (0 2). Thus, the first value of (-1 0) is not returned by CLIPS. This is like the case of deffunctions which only return the last action. One way of getting around this problem and returning both point values of the line is shown following.

```
CLIPS>
(defmessage-handler TRIANGLE find-line (?line)
    (create$
        (send
           (send (send ?self
                         (sym-cat "get-line"
                                  ?line))
                   get-point1)
           get-position)
        (send
           (send (send ?self
                         (sym-cat "get-line"
                                  ?line))
                   get-point2)
           get-position)))
CLIPS> (send [Triangle1] find-line 1)
(-1 0 0 2)
CLIPS>
```

Notice that the (create$) function was used to combine both point values into a single multifield value (-1 0 0 2) which was then returned.

Other Features

A number of other functions are useful with handlers. Some of these follow.

undefmessage-handler: Deletes a specified handler.

list-defmessage-handlers: Lists the handlers.

delete-instance: Operates on the active handler.

message-handler-existp: Returns TRUE if handler exists, else FALSE.

The **grouping functions** group COOL items into a multifield variable.

get-defmessage-handler-list: Groups the class names, message names, and types (direct or inherited).

class-superclasses: Groups all superclass names (direct or inherited).

class-subclasses: Groups all subclass names (direct or inherited).

class-slots: Groups all slot names (explicitly define or inherited).

slot-existp: Returns TRUE if the class slot exists, else FALSE.

slot-facets: Groups the specified slot facet values of a class.

slot-sources: Groups the slot names of classes which contribute to a slot in the specified class.

The **preview-send** function is useful in debugging since it displays the sequence of all handlers that *potentially* may be involved in processing a message. The reason for the term *potentially* is that shadowed handlers will not be executed if the shadower does not use call-next-handler or override-next-handler.

Handlers are arranged in a **message-handler precedence** which determines how they are called. The process of determining which handlers should be called and in what order is called the **message dispatch**. Every time a message is sent to an object, CLIPS arranges the message dispatch for the applicable handlers of that object, which can be viewed by the (preview-send) command. The applicable handlers are all handlers in all classes along the object's inheritance path that may respond to the type of message. Other functions useful with pattern matching objects by rules follow.

object-pattern-match-delay: Delay pattern matching of rules until after instances are created, modified, or deleted.

modify-instance: Modifies instance using slot overrides. Object pattern matching delayed until after modifications.

active-modify-instance: Change the values of the instance concurrent with object pattern matching with *direct-modify* message.

message-modify-instance: Change the values of the instance. Delay object pattern matching until all slots are changed.

active-message-modify-instance: Changes the values of the instance concurrent with object pattern matching using *message-modify*.

Questions and Answers

The best way to learn is by asking yourself questions; the best way to be sorry is by answering all of them.

In this chapter you'll learn how to pattern match on instances. One way is with rules. Also, CLIPS has a number of query functions to match instances. In addition, control facts and slot daemons can be used for pattern matching.

Object Lessons

One of the new features of Version 6.0 is the ability of rules to pattern match on objects. The following example shows how the value of the slot sound is pattern matched by a rule.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
    (multislot sound (default quack quack)))
CLIPS>
(make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS>
(make-instance [Dinky_Duck] of DUCK)
[Dinky_Duck]
CLIPS>
(defrule find-sound
    ?duck <- (object (is-a DUCK)
                     (sound $?find))
=>
    (printout t "Duck "
                (instance-name ?duck)
                " says " ?find crlf))
CLIPS> (run)
Duck [Dinky_Duck] says (quack quack)
```

```
   Duck [Dorky_Duck] says (quack quack)
   CLIPS>
```

The object-pattern conditional element object is followed by the classes and slots to be matched against. Following the is-a and the slot-name can be constraint expressions involving ?, $?, &, and |.

In addition, instance names can be specified for pattern matching. The following example shows how only one instance of the DUCK class is matched using the **name constraint**, **name**, of instances. Note that *name* is a reserved word and cannot be used as a slot name.

```
   CLIPS>
   (defrule find-sound
      ?duck <- (object (is-a DUCK)
                       (sound $?find)
                       (name [Dorky_Duck]))
   =>
      (printout t "Duck "
                 (instance-name ?duck)
                 " says " ?find crlf))
   CLIPS> (run)
   Duck [Dorky_Duck] says (quack quack)
   CLIPS>
```

Objects in the Database

Consider the following general type of problem. Given some instances, how many satisfy a specified condition? For example, shown following are the defclasses and definstances of Joe's Home showing the various types of sensors and the appliances connected to them. Notice that an abstract class DEVICE is defined since both SENSOR and APPLIANCE inherit common slots type and location.

```
   CLIPS> (clear)
   CLIPS>
   (defclass DEVICE (is-a USER)
      (role abstract)
      ; Sensor type
      (slot type (access initialize-only))
      ; Location
      (slot loc (access initialize-only)))
```

```
CLIPS>
(defclass SENSOR (is-a DEVICE)
    (role concrete)
    (slot reading)
    ; Min reading
    (slot min (access initialize-only))
    ; Max reading
    (slot max (access initialize-only))
    ; SEN. APP.
    (slot app (access initialize-only)))
CLIPS>
(defclass APPLIANCE (is-a DEVICE)
    (role concrete)
    ; Depends on appliance
    (slot setting)
    ; off or on
    (slot status))
CLIPS>
(definstances ENVIRONMENT_OBJECTS
    (T1 of SENSOR
        (type temperature)
        (loc kitchen)
        (reading 110) ; Too hot
        (min 20)
        (max 100)
        (app FR))
    (T2 of SENSOR
        (type temperature)
        (loc bedroom)
        (reading 10)  ; Too cold
        (min 20)
        (max 100)
        (app FR))
    (S1 of SENSOR
        (type smoke)
        (loc bedroom)
        (reading nil) ; Bad sensor nil reading
        (min 1000)
```

```
        (max 5000)
        (app SA))
    (W1 of SENSOR (type water)
        (loc basement)
        (reading 0)    ; OK
        (min 0)
        (max 0)
        (app WP))
    (FR of APPLIANCE
        (type furnace)
        (loc basement)
        (setting low) ; low or high
        (status on))
    (WP of APPLIANCE
        (type water_pump)
        (loc basement)
        (setting fixed)
        (status off))
    (SA of APPLIANCE
        (type smoke_alarm)
        (loc basement)
        (setting fixed)
        (status off)))
CLIPS>
```

Suppose the following questions or queries are asked. What are all the objects in the database? How are all the objects arranged? What are the relationships between objects? What are all the devices? What are all the sensors? What are all the appliances? Which sensor is connected to which appliance? Are there any sensors whose type is temperature? What sensors of type temperature have a reading between min and the max? An even more basic query is whether or not there are *any* sensors present.

The **query system** of COOL is a set of six functions that may be used for pattern matching an **instance-set** and performing actions. An *instance-set* is a set of instances, such as the instances of SENSOR. The instances in the instance-set may come from multiple classes that do not have to be related. In other words, the classes do not have to be from the same inheritance path.

The following list from the *CLIPS Reference Manual*, summarizes the predefined **query functions** that may be used for instance-set access.

any-instancep: Determines if one or more instance-sets satisfy a query.

find-instance: Returns the first instance-set that satisfies a query.

find-all-instances: Groups and returns all instance-sets which satisfy a query.

do-for-instance: Performs an action for the first instance-set which satisfies a query.

do-for-all-instances: Performs an action for every instance-set which satisfies a query as they are found.

delayed-do-for-all-instances: Groups all instance-sets which satisfy a query and then iterates an action over this group.

I'll Take Any

The **any-instancep** function is a predicate function that returns TRUE if there is an instance matching the pattern and FALSE otherwise. Shown following is an example of this query function used with the SENSOR and APPLIANCE classes and instances. The query function determines if there is *any* instance in the SENSOR class.

```
CLIPS> (reset)
CLIPS> (instances)
[T1] of SENSOR
[T2] of SENSOR
[S1] of SENSOR
[W1] of SENSOR
[FR] of APPLIANCE
[WP] of APPLIANCE
[SA] of APPLIANCE
For a total of 7 instances.
; Function returns TRUE because
; there is a SENSOR instance
CLIPS> (any-instancep ((?ins SENSOR)) TRUE)
TRUE
; Evaluation error—Bad!
; No DUCK class
CLIPS> (any-instancep ((?ins DUCK)) TRUE)
[PRNTUTIL1] Unable to find class DUCK.
CLIPS>
```

The basic format of a query function involves an **instance-set-template** to specify the instances and their classes, an **instance-set-query** as the boolean condition that the instances must satisfy, and **actions** to specify the actions to be taken. The predicate function **class-existp** returns TRUE if the class exists and FALSE otherwise.

The combination of instance name followed by the one or more **class restrictions** is called an **instance-set-member-template**. The query functions may generally be used like any other function in CLIPS.

There are two steps involved in trying to satisfy a query. First, CLIPS generates all the possible instance-sets that match the instance-set-template. Second, the boolean instance-set-query is applied to all the instance-sets to see which ones, if any, satisfy the query. Instance-sets are generated by a simple **permutation** of the members in a template, where the rightmost members are varied first. Note that a permutation is not the same as a **combination** because order matters in a permutation but not in a combination.

The function **find-all-instances** returns a multifield value of all instances which satisfy the query, or an empty multifield value for none. The **do-for-instance** query function is similar to find-instance except that it performs a single **distributed action** when the query is satisfied. The **do-for-all-instances** function is similar to the do-for-instance except that it performs its actions for every instance-set that satisfies the query.

Design Decisions

In contrast to rules which are only activated when their patterns are satisfied, deffunctions are explicitly called and then executed. Just because a rule is activated does not mean it will be executed. Deffunctions are completely procedural in nature because once called by name, their code is executed in a procedural manner, statement by statement. Also, no pattern matching involving constraints is used in a deffunction to decide if its actions should be executed. Instead, any arguments that match the number expected by the deffunction argument list will satisfy the deffunction and cause its actions to be executed.

The basic idea of deffunctions as named procedural code is carried to a much greater degree with **defgenerics** and the **defmethods** that describe their implementation. A defgeneric is like a deffunction but much more powerful because it can do different tasks depending on its argument constraints and types. The ability of a generic function to perform different actions depending on the classes of its arguments is called **overloading** the function name.

By proper use of operator overloading, it's possible to write code that is more readable and reusable. For example, a defgeneric for the "+" function can be defined with different defmethods. The expression,

```
(+ ?a ?b)
```

could add two real numbers represented by ?a and ?b, or two complex numbers, or two matrices, or concatenate two strings, and so forth depending if there is a defmethod defined for the argument classes. CLIPS does this by first recognizing the type of the arguments and then calling the appropriate defmethod defined for those types. A separate overloaded defmethod for "+" would be defined for each set of argument types except for the predefined system types such as real numbers. Once the defgeneric is defined, it's easy to reuse in other programs.

Any **named function** that is system defined or external can be overloaded using a generic function. Notice that a deffunction cannot be overloaded. An appropriate use of a **generic** function is to overload a named function. If overloading is not required, you should define a deffunction or an external function.

The syntax of defgenerics is very simple, consisting of just the legal CLIPS symbol name and an optional comment.

```
(defgeneric <name> [<comment>])
```

As a simple example of generic functions, consider the following attempt in CLIPS to compare two strings using the ">" function.

```
CLIPS> (clear)
CLIPS> (> "duck2" "duck1")
[ARGACCES5] Function > expected argument #1 to be of type
integer or float
CLIPS>
```

It's not possible to do this comparison with the ">" function because it expects NUMBER types as arguments.

However, it's easy to define a (defgeneric) which will overload the ">" to accept STRING types as well as NUMBER types. For example, if the arguments of ">" are of type STRING, the defgeneric will do a string comparison, letter by letter starting from the left until the ASCII codes differ. In contrast, if the arguments of ">" are of type NUMBER, the system compares the sign and magnitude of the numbers. The user-defined ">" for STRING types is an **explicit method**, while a system-defined or user-defined external function such as ">" for NUMBER type is an **implicit method**.

The technique of overloading a function name so that the method which implements it is not known until run-time is another example of dynamic binding. Any object reference of name or address may be bound at run-time in CLIPS to functions through dynamic binding also.

Some languages such as Ada have a more restrictive type of overloading in which the function name must be known at compile time rather than at run-time. The run-time *dynamic binding* is the least restrictive since methods can be created during execution by the (build) statement. However, you should be careful in using (build) since dynamically creating constructs is often hard to debug. Also, the resulting code may be difficult to verify and validate since you'll have to stop execution to examine the code. Dynamic binding is a characteristics of a true object-oriented programming language.

Following is an example of a defgeneric, ">", for STRING types and its method.

```
; Header declaration. Actually unnecessary
CLIPS> (defgeneric >)
CLIPS>
(defmethod > ((?a STRING) (?b STRING))
   (> (str-compare ?a ?b) 0))
; The overload ">" works correctly
; in all three cases.
CLIPS> (> "duck2" "duck1")
TRUE
CLIPS> (> "duck1" "duck1")
FALSE
CLIPS> (> "duck1" "duck2")
FALSE
CLIPS>
```

The (defgeneric) acts as a **header declaration** to declare the type of function being overloaded. It's not actually necessary to use a defgeneric in this case because CLIPS implicitly deduces the function name from the defmethod name, which is the first symbol following "defmethod". The header is a *forward declaration* that is necessary if the (defgeneric) methods have not yet been defined, but other code such as defrules, defmessage-handlers, and so forth refers to the (defgeneric) name.

Other Features

Compared to deffunctions, a method has an optional **method index**. If you don't supply this index, CLIPS will provide a unique index number among the methods for that generic function,

that can be viewed by the **list-defmethods** command. The **method body** can be printed using the **ppdefmethod** command. A method can be removed with an **undefmethod** function call.

The ranking of methods determines the **method precedence** of a generic function. It is the method precedence which determines the methods order of listing. Higher precedence methods are listed before lower precedence methods. The highest precedence method will also be tried first by CLIPS.

A **shadowed method** is one in which one method must be called by another. The process by which CLIPS picks the method with highest precedence is called the **generic dispatch**. For more information, see the *CLIPS Reference Manual*.

Support Information

Questions and Information

The URL for the CLIPS Web page is http://www.clipsrules.net.

Questions regarding CLIPS can be posted to one of several online forums including the CLIPS Expert System Group, http://groups.google.com/group/CLIPSESG/, the SourceForge CLIPS Forums, http://sourceforge.net/forum/?group_id=215471, and Stack Overflow, http://stackoverflow.com/questions/tagged/clips.

Inquiries related to the use or installation of CLIPS can be sent via electronic mail to support@clipsrules.net.

CLIPS Source Code and Executables

CLIPS executables and source code are available at http://sourceforge.net/projects/clipsrules/files.

Documentation

The CLIPS Reference Manuals and User's Guide are available in Portable Document Format (PDF) at http://www.clipsrules.net/?q=Documentation.

Expert Systems: Principles and Programming, 4th Edition, by Giarratano and Riley (ISBN 0-534-38447-1) comes with a CD-ROM containing CLIPS 6.22 executables (DOS, Windows XP, and Mac OS), documentation, and source code. The first half of the book is theory oriented and the second half covers rule-based programming using CLIPS. It is published by Course Technology.