Introduction
to
Symbolic Computing
(First Draft)

Robert L. Walton
walton@deas.harvard.edu

June 28, 1999

Cambridge, Massachusetts

Contents

List of Figures

Chapter 1

Symbolic Computing in LISP

In this chapter we introduce the notions of symbolic computing, abstract program description, and the LISP programming language.[1]

1.1 What is Symbolic Computing

One of the earliest applications of symbolic computation was to algebra. Consider what is necessary to program a computer to perform the computation:

$$
\begin{aligned}
\texttt{5 + 2 * x - 3} &\Longrightarrow \texttt{2 * x + 5 - 3}\\
&\Longrightarrow \texttt{2 * x + (5 - 3)}\\
&\Longrightarrow \texttt{2 * x + 2}
\end{aligned}
$$

where $\Longrightarrow$ means '**rewrites to**'. More specifically, '$E_1 \Longrightarrow E_2$' for two expressions E_1 and E_2 means that E_1 can be rewritten as (i.e. replaced by) E_2, without changing its essential meaning for the purposes of some calculation.

The above computation contains implicit parentheses, and keeping track of such would be a great bother to a computer program. So programming the computer would be easier if the parentheses were made explicit, as in:

[1] This book is based on experience teaching Computer Science 51 at Harvard, a course developed by Professors Harry Lewis, Henry Leitner, and Thomas Cheatham among others. This book precisely covers the first 40% of Computer Science 51 as taught by the author in the Harvard Summer School.

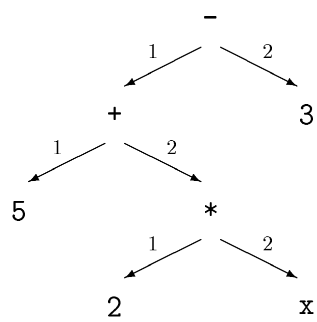
$$\begin{aligned} \texttt{((5 + (2 * x)) - 3)} &\Longrightarrow \texttt{(((2 * x) + 5) - 3)} \\ &\Longrightarrow \texttt{((2 * x) + (5 - 3))} \\ &\Longrightarrow \texttt{((2 * x) + 2)} \end{aligned}$$

Each subexpression in the above is just a short list of variables, numbers, operators, and smaller subexpressions, all surrounded by parentheses. The operator in the subexpression, e.g. `+` or `-` or `*`, is the second thing on each list in all the examples just given. But in expressions like `(x * (- y))` operators like '`-`' are the first thing in a list. It would be easier for the computer to find the operator if the operator were always first, as in:

$$\begin{aligned} \texttt{(- (+ 5 (* 2 x)) 3)} &\Longrightarrow \texttt{(- (+ (* 2 x) 5) 3)} \\ &\Longrightarrow \texttt{(+ (* 2 x) (- 5 3))} \\ &\Longrightarrow \texttt{(+ (* 2 x) 2)} \end{aligned}$$

where `(* 2 x)` is written instead of `(2 * x)`, `(+ 5 (* 2 x))` is written instead of `(5 + (2 * x))`, and so forth. The expressions in the computation just given are said to be in '**operator prefix form**', because the operator is always the first thing in each subexpression.

Now consider how the expressions in the last computation might be stored in computer memory. They could be stored as character strings, but shuffling characters in character strings is very expensive computationally (it takes the computer a long time to do). A more efficient way to store expressions is as '**labeled graphs**', such as:



which represents the expression `(- (+ 5 (* 2 x)) 3)` (for which the non-prefix form is `((5 + (2 * x)) - 3)`). A labeled graph consists of nodes and arrows.[2] The nodes

[2]Actually such a graph is a '**directed**' labeled graph, because the arrows have a direction. A labeled graph in which line segments appear instead of arrows is called '**undirected**'.

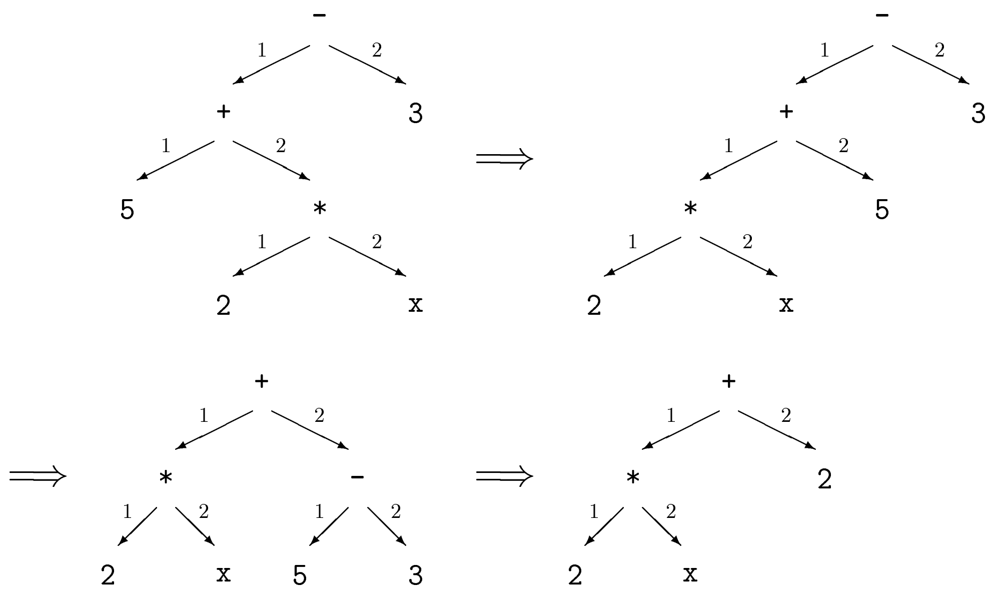
in this graph have labels `-`, `+`, `3`, `5`, `*`, `2`, and `x`. The arrows have labels 1 and 2 to denote the first or second operand of an operator. The word '**graph**' is being used here as computer scientists use it, and not as most mathematicians, statisticians, or business people would use it.

This particular graph is called a '**tree**' because it looks like a tree drawn upside down. Computer scientists usually draw trees upside down, with the '**root**' node of the tree at the top of the picture. In our example the root node has the label '`-`'.

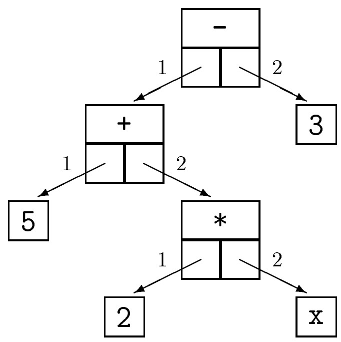
A tree is used to represent an expression by drawing a node for each operator and drawing downward arrows from that node to the operands of the operator. Each node is labeled by its operator, and the arrows are labeled 1, 2, 3, ..., according to whether the arrow corresponds to the first, second, third, ... operand of the operator.

A node is also drawn for each number or variable, with no arrows pointing from such a number or variable node. Nodes with no arrows pointing from them are called '**leaves**' of the tree. In our examples the leaves have the labels '`5`', '`2`', '`x`', and '`3`'.

Writing expressions in the form of trees, the computation we are asking the computer to perform is:

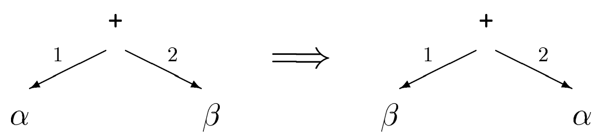


Inside a computer a node is represented as a small contiguous block of memory, and an arrow is represented by storing a pointer in the block of the node that is the origin of the arrow. This pointer points to the node that is the destination of the arrow. (A pointer is an address: a pointer to a memory location is the address of the memory location.) Thus we could picture the first expression tree above as:



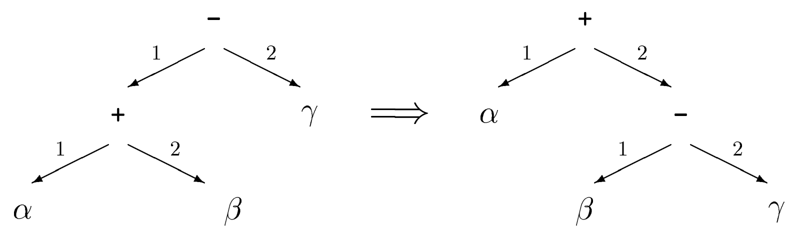
Here each node is represented by a box, which symbolizes the contiguous block of computer memory that represents the node. Each operator node box is divided into three parts, which represent places in the block to store the node label and the two pointers that represent the arrows originating at the node. Each pointer is represented as an arrow from the place the pointer is stored in memory to the node in memory the pointer points at. And each leaf node is represented by a box which has only one part that stores the node label.

Thus to perform the first expression rewrite above, the computer merely needs to exchange the pointers in the `+` node. We can say that this rewrite applies the rule `(+` α β`)`$\Longrightarrow$ `(+` β α`)` where α and β denote arbitrary expressions. And we can express this '**rewrite rule**' in the 'language of labeled graphs' as:



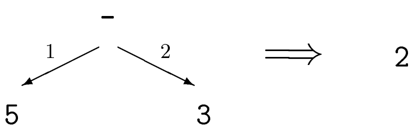
where now α and β represent expression trees.

To perform the second rewrite above, we apply the rewrite rule `(- (+` α β`)` γ`)`$\Longrightarrow$ `(+` α `(-` β γ`))`, where α, β, and γ denote arbitrary expressions. We can express this rewrite rule in the language of labeled graphs as:



where now α, β, and γ represent expression trees. The rule is applied by moving pointers around in the `+` and `-` nodes, and changing the root of the tree from the `+` node to the `-` node.

The third and last rewrite above simply applies the rewrite rule `(- 5 3)`$\Longrightarrow$`2`, or in the language of labeled graphs:



This requires the computer to make a new node labeled `2`, to dispose of the nodes labeled `-`, `5`, and `3`,[3] and to change any pointer directed at the discarded node labeled `-` to point at the new node labeled `2`.

It turns out that computers can perform computations with expressions much more efficiently if the expressions are stored as trees rather than as character strings, because computers can read expressions one node at a time instead of one character at a time, and because computers can change expressions by writing one pointer at a time instead of one character at a time. There are usually many more characters in an expression than there are nodes and arrows in the expression tree, so the character form of the expression is much less efficient to compute with.

In this book we will assume that symbolic computing means computing with labeled graphs.

There are many other labeled graphs that occur in computing. One of these is the directory tree of a file system. Others that are discussed later in this book are the object graph that represents the state of computer memory in any program, the call tree that represents the execution history of any program, and the different kinds

[3]Such disposal of nodes that are no longer in use is called '**garbage collection**'.

of search trees that represent the possible paths of a search. As this book is just an introduction, this list of labeled graphs is just a fraction of those that occur in computing.

We do not mean to imply that symbolic computing is really restricted to computing with labeled graphs. For example, computing with character strings by matching patterns is symbolic computing, but we do not cover character string pattern matching in this book. For another example, relational databases are the basis for a kind of symbolic computing that is also not covered in this book. These and other symbolic computing topics are very useful for particular computer users who apply computers in particular ways, but they are not as central to computation as labeled graphs.

Therefore, as our concern is the introductory education of computer scientists, we stick to the more central topic of computing with labeled graphs.

This first chapter is an introduction to labeled graphs, and to the LISP programming language as used for storing and computing with labeled graphs.

1.2 Abstract Description of Programs

One of the main tasks of both computer scientists and computer programmers is to read and write abstract descriptions of programs. An **abstract description** of a computer program is just any description of the program that is several times easier to read or write than the code of the program. Usually abstract descriptions are also several times shorter than the total code of the program.

A main goal of this book is to teach students how to read abstract descriptions of computer programs. A second goal is to make a beginning at teaching students how to write such descriptions.

Abstract descriptions of computer programs are often called called **specifications** or **designs**.

In the commercial programming world, abstract descriptions have three common uses.

First, they are used to describe the program to any potential buyer of the the program.

Second, abstract descriptions are used to avoid errors in which a program tries to do the wrong thing. That is, the algorithm the programmer is trying to implement is defective, but the code correctly implements this incorrect algorithm. These errors will most likely be found when a user of the program understands what the program is trying to do.

Third, on large projects in which many different programs written by different programmers must fit together, an abstract description of a program is often used to find errors arising because the programmer who is writing the program and other programmers whose code uses the program have different ideas of what the program does. These errors will most likely be found when the writer and users of a program describe concisely what the program is trying to do.

In this book we use several tools for describing programs abstractly. Two have already been introduced: rewrite rules, and labeled graphs. We will use different kinds of labeled graph, for example the call tree and the object graph. Another tool we use is the context free grammar.

It is mainly by using these tools that this book teaches the basics of abstract program description.

Exercise 1.2.1 Try to describe the computation `(x + 5) - 3` $\Longrightarrow$ `x + (5 - 3)` <u>without</u> using any notion of rewrite rule or labeled graph.

Exercise 1.2.2 Give a brief but precise description of the structure of information in the file system of your favorite operating system using labeled graphs. In a file system graph, what kind of nodes are leaves, and what kind are non-leaves?

1.3 An Introduction to LISP

The **LISP** programming language was one of the earliest programming languages specialized for symbolic computing, and has been the most successful such language to date. It is often used by researchers in artificial intelligence and natural language understanding. The language of the commercial symbolic computing system Mathematica^{TM}can be viewed as a variant of LISP, just as Mathematica itself is a descendant of an earlier research system named MAXSYMA that was written in LISP.

LISP can be both interpreted and compiled, and a LISP program can be a mixture of interpreted and compiled code. Generally speaking, it is easier to program in an interpreted language than in a compiled language, because debugging is easier, since it is possible to modify large programs written in an interpreted language more quickly because recompilation is not required.

One very powerful idea is that of combining the LISP and **C** programming languages to write very large programs whose parts that must run fast are written in C and whose parts that can run slowly are written in LISP. Typically at least half of such a large program can be written in LISP, and typically this is the half that is most likely to change during the course of program development, since code that must perform quickly often does things that are conceptually simpler and less likely to change.

This idea for combining LISP and C has been used successfully for some large commercial pieces of software, such as the Interleaf™publishing system, and for some large and widely used pieces of free software, such as the EMACS editor.

One of the most successful pieces of software for running web stores is written in LISP, and is currently running the Yahoo™web store.

At the current time (1999) it is unclear whether **JAVA** will ever replace LISP, but it looks like it may fail to do so. The strong typing system of JAVA makes it more difficult to design very large complex programs, like those just mentioned. Although JAVA contains a weakly typed subset analogous to LISP,[4] that subset has not been a success so far in replacing LISP, perhaps for efficiency or ease-of-use reasons, or perhaps just due to inertia. Also JAVA does not have the LISP ability to treat code as data, which is sometimes important.

LISP has been very successfully standardized. The standard is called **COMMONLISP**, and there are good free implementations, such as **CLISP**[5].

We will introduce and use a very small subset of COMMONLISP as the programming language of this book. However, we do not completely describe all the COMMONLISP functions we use in this book, and the reader will need to occasionally reference COMMONLISP documentation, preferably a good textbook such as [Gra96].[6]

We furthermore use COMMONLISP in a very particular way. We stick almost completely to two styles of programming called '**functional programming**' and '**recursive programming**'. Functional programming means refusing to use variables that change their values after they are created. Recursive programming means refusing to use loops, and instead using recursive functions.

We have three reasons for sticking to a recursive functional programming style in this book. First, it is the style that lets us use the smallest possible subset of the LISP language, so we can spend less time learning the language and more time learning other concepts. Second, it is the style that best matches good abstract descriptions

[4]Consider the `Integer`, `String`, and `Vector` classes.

[5]`http://clisp.cons.org` CLISP has been used to run the examples in this book.

[6]`www.harlequin.com/books/HyperSpec/` has ANSI standard COMMONLISP documentation that if downloaded to reduce network traffic makes a good online reference.

of programs using rewrite rules, and such descriptions are among the best abstract descriptions for the kinds of symbolic computing that we treat in this book.

And third, both functional and recursive programming are styles that young computer scientists should learn early and practice often; and the study of symbolic computing is one of the best places to learn these styles. There are many other places to learn the other good styles of programming, so their omission from this book should not harm the reader.

1.3.1 Executing COMMONLISP

A COMMONLISP interpreter is a program that behaves like a calculator. If you start such a program you can hold the following dialog with it:

```
> (+ 2 5)
7
>
```

Here the computer types '`> `', and waits for the user to type an expression. Next the user types '`(+ 2 5)`' and the RETURN (or ENTER) key. The computer evaluates the expression the user just typed, and types the resulting value, '`7`', on a line by itself. Then the computer begins a repetition of this process by typing another line beginning with '`> `', and waiting for the user to type another expression.

The interpreter is running what is called a '**read-eval-print**' loop. First an expression is read, then it is evaluated, and lastly the result of evaluation is printed.

In this particular read-eval-print loop, the read part itself consists of two subparts. The first prints a **prompt**, '`> `', and the second reads an expression. Other read-eval-print loops may read expressions from files and not print any prompt.

Notice that LISP expects all parentheses to be explicit and all operators to be in the prefix position, before their operands.

A very good way to learn an interpreted language such as LISP is to start an interpreter and then, as you read a book describing the language, use the interpreter to try out each new language concept as you read about that concept.

In this book we will commonly show output produced by a different read-eval-print loop. This different loop reads from an input file, and prints each line read. When it finishes reading an expression, it prints a line containing '`--->`' and then prints the result of evaluating the expression on another line. If the input file contains:

```
;; This is a comment line.
(+ 2 5)
```

then the printed output will be:

```
;; This is a comment line.
(+ 2 5)
--->
7
```

As we can see from this example, the LISP interpreter treats lines beginning with a semi-colon ';' as a **comment**, and ignores these lines. In fact, any 'unquoted and unescaped' semi-colon in an input line begins a comment that lasts until the end of the line. Later we will define escaped characters, but we never use or discuss quoted characters in this book.

Exercise 1.3.1.1 Use a LISP interpreter to find out what the interpreter outputs for each of the following input lines. In a few cases you can find the answers in the figures of the rest of this chapter.

(a)	`(- 9 2)`	
(b)	`(/ 8 6)`	
(c)	`(+ 1 2 3 4 5)`	
(d)	`'(+ 1 2 3 4 5)`	(note the quote `'`)
(e)	`(sq 8)`	(if done before (f))
(f)	`(defun sq (x) (* x x))`	
(g)	`(sq 8)`	(if done after (f))
(h)	`(+ x 5)`	(if done before (i))
(i)	`(setf x 101)`	
(j)	`(+ x 5)`	(if done after (i))
(k)	`(setf y x)`	(if done after (i))
(l)	`y`	(if done after (k))
(m)	`(setf y 'x)`	(note the quote `'`)
(n)	`y`	(if done after (m))

1.4 Pure LISP, Arithmetic, and Rewrite Rules

In this section we will describe programming in LISP with numbers and recursive functions.

1.4.1 LISP Numbers and Arithmetic

COMMONLISP understands three kinds of numbers:

Integers:	`1    9    -55    12345678901234567890`
Floating Point Numbers:	`1.0    0.1    -0.09    3e6`
Ratios:	`1/3    57/64    -2/3    1234567890987654321/13`

Integers can be as long as you like, unlike in most other programming languages.

Floating point numbers are similar to those of other programming languages. In this book floating point numbers have up to 16 meaningful decimal digits, but a COMMONLISP interpreter is typically set up to provide floating point numbers with only 8 meaningful decimal digits.[7] We will make little use of floating point numbers in this book.

Ratios are just fractions. They are reduced so that the numerator and denominator have no common divisor, and the denominator is greater than 1. An integer can be thought of as if it were a ratio with denominator 1, but strictly speaking integers are not ratios.

You can type in a ratio like `8/10` whose numerator and denominator have a common factor, but it will be stored and printed as the ratio with the common factor removed, so in this case `4/5` is stored and printed.

Figure 1.1 gives examples inputting simple numbers into a COMMONLISP interpreter. Figure 1.2 gives examples inputting arithmetic expressions which we will now discuss.

All arithmetic expressions are explicitly parenthesized and have the operator in the prefix position. Even an expression with only a single operator must have parentheses surrounding the expression.

Expressions with exactly two operands and with the operator `+`, `-`, or `*` (this last denotes multiplication) do the obvious thing. So do expressions with the division operator `/` and two floating point number operands.

If the two operands to `+`, `-`, `*`, or `/` are integers or ratios, the answer is an integer or a ratio. This is known as **exact arithmetic**: the answer is an exact number

[7] The number of meaningful digits in a floating point number is called its **precision**. The precision of a COMMONLISP interpreter is the precision assumed when floating point numbers are read, and is controlled by the value of the `*read-default-float-format*` variable, as described in books on COMMONLISP.

```
;; Integers:              ;; Floating     ;; Ratios:
                          ;; Point
9                         ;; Numbers:     1/3
--->                                      --->
9                         1.0             1/3
                          --->
-55                       1.0             -57/64
--->                                      --->
-55                       -0.09           -57/64
                          --->
12345678901234567890      -0.09           1234567890987654321/13
--->                                      --->
12345678901234567890      3e6             1234567890987654321/13
                          --->
                          3000000.0       8/10
                                          --->
                                          4/5
```

Figure 1.1: Examples of Numbers

with no rounding. Thus `(/ 8 6)`⟹`4/3` in COMMONLISP, whereas in many other programming languages the result would be `1`.

If <u>any</u> operand is a floating point number, COMMONLISP produces a rounded floating point number result. This is known as **inexact arithmetic**. An example is `(/ 2.0 3.0)`⟹`0.6666666666666666`.

What if you want rounding? Then you have to use a function such as the one named `floor`. This works just like `/` except that it rounds its result to the largest integer not greater than the exact result. Thus `(floor 8 6)`⟹`1`.

We called `floor` a function and `/` an operator, but what is the difference in COMMONLISP? The answer is there is none. COMMONLISP actually does not have any operators; it only has **functions**. What we call **operators**, `+`, `-`, `*`, and `/`, are

```
;; Arithmetic:

(+ 9 5)
--->
14

(- 9 5)
--->
4

(* 9 5)
--->
45

(/ 9.0 5.0)
--->
1.8

(+ (* 9 5) (- 9 5))
--->
49

;; Exact Arithmetic

(* 2/3 1/2)
--->
1/3

(/ 2 3)
--->
2/3

(* 123456789 987654321)
--->
121932631112635269

;; Inexact Arithmetic

(/ 2.0 3.0)
--->
0.6666666666666666
```

```
;; Divide and Round Down
;; (toward minus infinity)
;; to an Integer:

(floor 17 5)
--->
3

(mod 17 5)
--->
2

(floor -17 5)
--->
-4

(mod -17 5)
--->
3

;; Special Cases:

(+)
--->
0

(*)
--->
1

(* 5)
--->
5

(+ 1 2 3 4 5)
--->
15

(* 1 2 3 4 5)
--->
120
```

Figure 1.2: Examples of Arithmetic Expressions

actually just the names of functions in COMMONLISP. Of course they look like the names of operators in algebra or in other programming languages.[8]

Thus `(floor 8 6)` is a COMMONLISP **function call**. In algebra or most other programming languages such a function call would be written as '`floor (8, 6)`'. But not in LISP, where function calls consist of the function name followed by the function arguments, all surrounded by parentheses, without any commas.

The absence of commas works because any argument that itself contains a function call is surrounded by parentheses, as is `(* 4 2)` in:

```
(floor (* 4 2) 6)
```

The function `floor` always rounds towards negative infinity, meaning that it moves the exact result toward minus infinity until it gets to an integer. As a consequence, `(floor -4 3)`$\Longrightarrow$`-2`, and not `-1`. Division as taught in the American school system would return `-1`, but `floor` is different.

There is a remainder function that goes with `floor`, and its name is `mod`, which is short for '**modulo**'. This function is defined so that:

```
x  =  y * (floor x y)  +  (mod x y)
```

is always true for any numbers `x` and `y` (this equation is a mixture of standard algebraic notation and two COMMONLISP style function calls). Thus `(mod -4 3)`$\Longrightarrow$`2` because `-4 = 3 * -2 + 2`. Note that because `floor` always rounds in the negative direction, `mod` always returns a result equal to `0` or between `0` and `y` (`y` can be negative).

If `x` and `y` are integers or ratios, the result of `floor` is an integer and the result of `mod` is an integer or ratio, and both results are exact. If either `x` or `y` is a floating point number, the result of `floor` is still an integer, but the result of `mod` is a floating point number that may be rounded and inexact.

The result of `(mod x y)` is read '`x` modulo `y`'. The concept of numbers modulo other numbers is very important in computer science.

The operators `+` and `*` can also take less than 2 or more than 2 operands. These operators simply add or multiply all their operands together. If given zero operands, `+` returns `0` while `*` returns `1`. That is, `(+)`$\Longrightarrow$`0` and `(*)`$\Longrightarrow$`1`.

[8]Because LISP treats operators just like functions, it is easier for a LISP program to perform computations on algebraic expressions.

The operator `-` can also take only one argument and return the negative of its single argument.

Exercise 1.4.1.1 Show how the following LISP expressions are rewritten into numbers when they are evaluated by a LISP interpreter. E.g.:

`(/ (+ 3 4) 3.0)` $\Longrightarrow$ `(/ 7 3.0)` $\Longrightarrow$ `2.333333333333333`

(a) `(+ 5 6)`
(b) `(+ 5 6.0)`
(c) `((* 5 6) + 9 )`
(d) `(- (* 5 6))`
(e) `(floor (* 5 7) (- 4 1))`
(f) `(mod (* 5 7) (- 4 1))`

1.4.2 Rewrite Rules and Function Definitions

Suppose we want to define in LISP a function named `circle-area` that obeys the rewrite rules:

`(circle-area` n`)` $\Longrightarrow$ `(* 3.1415` n n`)` where $n \in$ number

Here $\Longrightarrow$ means '**rewrites to**' and $\in$ means '**is a**'. What we just described is not just a single rewrite rule, but a whole set of rewrite rules, one for each possible number n. Some of these are:

```
(circle-area 2)    ⟹  (* 3.1415 2 2)
(circle-area 0.5)  ⟹  (* 3.1415 0.5 0.5)
(circle-area 9/2)  ⟹  (* 3.1415 9/2 9/2)
```

In LISP we can define the function `circle-area` by writing the code:

```
(defun circle-area (n) (* 3.1415 n n))
```

This is called a **function definition** in LISP. It has the form of a function call on the function named `defun`, in which the first operand (e.g., `circle-area`) is the name of the function being defined, the second operand (e.g., '`(n)`') is a list of argument names, and the third operand (e.g., '(* 3.1415 n n)') is the expression that a call to the named function rewrites to after the argument names have been replaced by their values.

Just for comparison, the same function would be defined in the C programming language by:

```
float circle_area (float n) { return 3.1415 * n * n; }
```

Figure 1.3 gives the results of inputing into a LISP interpreter a file containing a definition and example uses of the `circle-area` function. The second example use, for example, gives the result of the computation:

$$\texttt{(circle-area 0.5)} \Longrightarrow \texttt{(* 3.1415 0.5 0.5)} \Longrightarrow \texttt{0.785375}$$

The definition of `circle-area` in Figure 1.3 is given in two forms: first as rewrite rules in comments (lines beginning with ';' are comments lines), and second as LISP function definition code. For more complex functions, the rewrite form is likely to say what we mean more directly and compactly, so we always prefer that form when describing functions in this book. Thus the rewrite form serves as our means of abstractly describing functions. Since files can only contain simple ASCII[9] characters, '===>' and 'is a' are used in the file in place of $\Longrightarrow$ and $\in$.

The evaluation of a function definition causes the interpreter first to remember the definition of the function, and second to return the name of the function as the 'value' of the function definition itself.

As can be seen in Figure 1.3, the returned name is capitalized (e.g., `CIRCLE-AREA`). Later we will see that a function name in LISP is something called a LISP symbol, which is roughly like an English word, but unlike English, LISP is '**case-insensitive**', which means LISP does not distinguish between the upper and lower case versions of a letter. E.g., LISP does not distinguish between 'c' and 'C'. When a LISP interpreter reads a symbol, it immediately translates all 'unescaped' lower case letters to upper case, and stores the upper case version of the symbol in the memory of the interpreter. Later we will define escaped characters.

Thus `circle-area`, `CIRCLE-AREA`, `Circle-Area`, and `circlE-aReA` are all the same function name in LISP. We usually type `circle-area` when inputting this function

[9]**ASCII** stands for American Standard Code for Information Interchange and denotes a standard set of characters understood by almost all modern computers.

```
;; Rewrite Rules

;; (circle-area n) ===> (* 3.1415 n n)
;;
;; where n is a number
;;
(defun circle-area (n) (* 3.1415 n n))
--->
CIRCLE-AREA

(circle-area 1)
--->
3.1415

(circle-area 0.5)
--->
0.785375

(circle-area 9/2)
--->
63.61537500000001


;; (square n) ===> (* n n)
;;
;; where n is a number
;;
(defun square (n) (* n n))
--->
SQUARE

(square 1/3)
--->
1/9

(square 3)
--->
9
```

Figure 1.3: Examples of Function Definitions

name to the LISP interpreter or to a file, because it is easier to type lower case letters. The LISP interpreter outputs this function name as `CIRCLE-AREA` because it converts unescaped letters from lower to upper case on input.

Figure 1.3 also contains the rewrite rules, LISP function definition code, and example uses of the `square` function that simply returns the square of its input argument.

Exercise 1.4.2.1 Define functions to:
 (a) Find the area of a rectangle.
 (b) Find the average of weight of Bob, Bill, and Boris.

Exercise 1.4.2.2 In words, what do the following functions do?
 (a) `(defun foo (diameter) (* 3.1415 diameter))`
 (b) `(defun bar (x) (* (mod x 2) x))`

1.4.3 True/False, Relations, and Predicates

LISP uses special values to represent **true** and **false**. False is represented by the symbol '`nil`', and true is represented by any other value, but if no particular other value is handy, then true is represented by the symbol '`t`'.

We can write this information in the shorthand:

false	::=	`nil`
true	::=	any-non-`nil`-value [default '`t`']

where '::=' is a symbol often used by computer scientists that means 'is' syntactically. Note that '::=' differs from $\in$ which means 'is a' set theoretically, in that `nil` in the first equation above represents itself and not a set. So this equation says that the set of possible inputs the LISP interpreter that represent 'false' is equal to the set whose only element is `nil`.

The second equation above also uses the notion of a **default value**, which is just a value that is used for some purpose when no other value is supplied for that purpose.

It may seem unusual to represent false by the symbol '`nil`'. Later we will see that `nil` also represents the empty list, so its use to denote false makes it easy to test whether a list is empty. In English 'nil' means 'nothing', or after a mild stretch, 'empty'.

The COMMONLISP functions `=`, `/=`, `>`, `<`, `>=`, and `<=` compute relations between numbers. We will call these functions '**relation operators**'. For example,

```
;; Two Operand
;; Relations

(< 5 6)
--->
T

(<= 5 6)
--->
T

(> 5 6)
--->
NIL

(>= 5 6)
--->
NIL

(= 5 6)
--->
NIL

(/= 5 6)
--->
T
```

```
;; Many Operand
;; Relations

(< 5 6 7 8)
--->
T

(< 5 9 7 8)
--->
NIL

(= 5 5 5 5)
--->
T

(= 5 5 6 6)
--->
NIL

(/= 5 6 4)
--->
T

(/= 5 6 5)
--->
NIL
```

```
;; Predicates

(evenp 3)
--->
NIL

(evenp 4)
--->
T

(oddp 3)
--->
T

(oddp 4)
--->
NIL
```

Figure 1.4: Examples of Relations and Predicates

```
(= 5 5)  ⟹  t
(= 5 6)  ⟹  nil
```

because `=` tests numbers for equality, `5` equals itself and '`t`' denotes true, while `5` does not equal `6` and `nil` denotes false.

`/=` means 'is not equal to', `>=` means 'is greater than or equal to', and `<=` means 'is less than or equal to'.

Examples using relation operators are given in Figure 1.4.

These relation operators can also be used with more than two operands. When this is done, they mean that the relation holds between every ordered pair of operands. Thus `(< 5 6 7)` is true because `(< 5 6)`, `(< 6 7)`, and `(< 5 7)` are all true; but `(/= 5 6 5)` is false because although `(/= 5 6)`, `(/= 6 5)` are both true, `(/= 5 5)` involving the first and third operand is false.

COMMONLISP also provides functions called '**predicates**' that test various things. Two that we will use are `evenp` and `oddp` that test whether an integer is even or odd. See Figure 1.4 for examples. By convention, the letter '`p`' is often placed at the end of the name of a LISP predicate function.

1.4.4 Conditional Expressions

In order for computers to have any intelligence at all, it is necessary for them to be able to do different things in different situations. The heart of this matter for LISP is conditional expression evaluation, in which different expressions are evaluated if different conditions are satisfied.

In LISP the '`cond`' expression provides for conditional expression evaluation. An example of a `cond` expression is:

```
(cond ((oddp 8)  (+ 2 3))
      ((evenp 8) (- 5 2)))   ⟹  (- 5 2)  ⟹  3
```

This first evaluates `(oddp 8)`, and because the value is false, skips evaluation of `(+ 2 3)`. Then `(evenp 8)` is evaluated, and because the value is true, `(- 5 2)` is evaluated and its value becomes the final value of the `cond` expression.

In general, a `cond` expression has the form[10] and function:

[10] COMMONLISP also has a more general form of `cond` expression: see any COMMONLISP text book.

```
;; (my-max m n) ===> m          if (>= m n)
;;              ===> n          otherwise
;;
;; where m, n are numbers
;;
(defun my-max (m n)
  (cond ((>= m n) m)
        (t       n)))
--->
MY-MAX

(my-max 2 9.7)
--->
9.7

(my-max 3/4 1/2)
--->
3/4
```

Figure 1.5: Example of a `COND` Expression

$$\begin{array}{lll} \texttt{(cond } (e_1\ v_1) & \Longrightarrow\ v_i & \text{if } e_i \text{ is the first of } e_1,\, e_2,\, \ldots \text{ that evaluates to true} \\ \qquad\quad (e_2\ v_2) & & \\ \qquad\quad \ldots\texttt{)} & \Longrightarrow\ \texttt{nil} & \text{if none of } e_1,\, e_2,\, \ldots \text{ evaluates to true} \end{array}$$

Thus a `cond` expression consists of a sequence of pairs of expressions ‘(e_1 v_1)’, ‘(e_2 v_2)’, ... To evaluate the `cond` expression, the expressions e_1, e_2, ... are evaluated in order until one, call it e_i, has a true value. Then the corresponding expression v_i is evaluated, and its value is returned as the resulting value of the `cond` expression.

If none of the expressions e_1, e_2, ... evaluate to true, the value of the `cond` expression is `nil`.

An example using the `cond` expression is:

```
(defun max (m n)
  (cond ((>= m n) m)
        (t        n)))
```

This function returns the maximum of `m` and `n`. If `m` is greater than equal to `n`, then `m` is returned. Otherwise `n` is returned.

'`t`' is often used at the end of a `cond` expression to mean 'otherwise'. In such a use, it is an expression whose value is always true, so if all the other previous expressions that are tested evaluate to false, then '`t`' will be evaluated, and will evaluate to true. Thus it means: 'otherwise if all the previous tests are false'.

Figure 1.5 illustrates the reaction of the LISP interpreter to the above definition of the `max` function. Because COMMONLISP already defines a function named `max`, the name of the function in this example has been changed to `my-max`.

Exercise 1.4.4.1 Pretend that the functions `oddp` and `evenp` do not exist and write a function definition for `oddp`.[11]

1.4.5 Recursion

One of the most important methods of computer programming is to reduce a computational problem to some previously solved computational problem. A subtle variation on this is to reduce a computational problem to the same problem but with smaller input. This last is called **recursion**.[12]

For example, consider the problem of computing the **factorial** function, $n!$, which multiplies together all the integers from 1 through n. Thus, for example, $5! \Longrightarrow 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$.

A solution to the problem of computing the factorial function is expressed by the rewrite rules:

$$\begin{array}{lcll} 0! & \Longrightarrow & 1 & \\ n! & \Longrightarrow & n \cdot (n-1)! & \quad \text{if } n > 0,\ n \in \text{integer} \end{array}$$

These rewrite rules can be translated into LISP notation as:

[11] Hint: think `mod`.

[12] Recursion is borrowed from mathematics where it is called **induction**.

```
;; (factorial 0) ===> 1
;; (factorial n) ===> (* n (factorial (- n 1)))
;;                        if (> n 0)
;; where n is an integer
;;
(defun factorial (n)
  (cond ((= n 0) 1)
        ((> n 0)
         (* n (factorial (- n 1))))))
--->
FACTORIAL

(factorial 5)
--->
120

(trace factorial)
--->
;; Tracing function FACTORIAL.
(FACTORIAL)

(factorial 5)
--->

1. Trace: (FACTORIAL '5)
2. Trace: (FACTORIAL '4)
3. Trace: (FACTORIAL '3)
4. Trace: (FACTORIAL '2)
5. Trace: (FACTORIAL '1)
6. Trace: (FACTORIAL '0)
6. Trace: FACTORIAL ==> 1
5. Trace: FACTORIAL ==> 1
4. Trace: FACTORIAL ==> 2
3. Trace: FACTORIAL ==> 6
2. Trace: FACTORIAL ==> 24
1. Trace: FACTORIAL ==> 120
120
```

Figure 1.6: Example of a Recursive Function

```
(factorial 0)   ==>   1
(factorial n)   ==>   (* n (factorial (- n 1)))
                         if (> n 0)
    where n is an integer
```

and this in turn can be translated into the LISP function definition:

```
(defun factorial (n)
  (cond ((= n 0) 1)
        ((> n 0)
         (* n (factorial (- n 1))))))
```

Figure 1.6 displays an example that inputs this definition into a LISP interpreter. This figure also gives a trace of an execution of this function that will be explained in Section 1.4.7 below.

Exercise 1.4.5.1 Write a recursive function that counts the number of ways of choosing a team of `m` players from `n` people, for integers $\mathtt{m} \geq 1$ and $\mathtt{n} \geq 1$. Your function must be recursive in a natural way, and in particular you must <u>not</u> use mathematical theory to get the answer by invoking the `factorial` function.[13]

Exercise 1.4.5.2 Write a recursive function that returns '`t`' if there is a way of putting `m` pigs in `n` pens so that each pen holds a non-zero odd number of pigs, for integers $\mathtt{m} \geq 1$ and $\mathtt{n} \geq 1$. If there is no such way the function returns `nil`. Your function must be recursive in a natural way, and in particular you must <u>not</u> use mathematical theory to get the answer by invoking a simple non-recursive computation.[14]

1.4.6 Rewrite Rules and Meta-Rules

Above we introduced the **rewrite rules**:

```
(factorial 0)   ==>   1
(factorial n)   ==>   (* n (factorial (- n 1)))
                         if (> n 0)
    where n is an integer
```

[13]Hint: if $\mathtt{n} = 1$ or $\mathtt{m} = 1$ its easy, and if $\mathtt{n} > 1$ and $\mathtt{m} > 1$, consider some person P, and add the number of ways of choosing a team including person P to the number of ways of choosing a team excluding the person P.

[14]Hint: if $\mathtt{n} = 1$ its easy, and if $\mathtt{n} > 1$, your best chance comes from putting just one pig in the first pen.

Actually, the second of these rules is not a rewrite rule per se, but is instead a **rewrite meta-rule** that represents an infinite number of rewrite rules, one for each value of `n` that satisfies the given conditions, namely that `n` is an integer and `(> n 0)`. Some of the infinite number of rules that are represented by this one meta-rule are:

```
(factorial 1)  ==>  (* 1 (factorial (- 1 1)))
(factorial 2)  ==>  (* 2 (factorial (- 2 1)))
(factorial 3)  ==>  (* 3 (factorial (- 3 1)))
(factorial 4)  ==>  (* 4 (factorial (- 4 1)))
```

In this rewrite meta-rule:

```
(factorial n)   ==>   (* n (factorial (- n 1)))
                        if (> n 0)
   where n is an integer
```

`n` is a **meta-variable**. It is <u>not</u> a LISP variable. Later when we deal with symbolic programming we will see the difference very clearly.

Because `n` is a meta-variable, `(factorial n)` is <u>not</u> a LISP expression. Instead, because it contains a meta-variable, it is a **meta-expression**. It becomes a LISP expression only after a LISP expression satisfying the given conditions is substituted for `n`. Thus after `2` is substituted for `n`, we get `(factorial 2)` which is a valid LISP expression.

Similarly, only after substitution for meta-variables does a rewrite meta-rule become a rewrite rule.

The two sides of a LISP rewrite rule are LISP expressions, and the rewrite rule means that its left side LISP expression can be replaced by its right side LISP expression for the purposes of computation.

In the rest of the book we will '**abuse language**' and call a meta-expression an expression and a meta-rule a rule. However, we will always call a meta-variable a meta-variable. With this loose language, a meta-expression or meta-rule is just an expression or rule that contains a meta-variable.

It is important to be able to identify the meta-variables in a rewrite rule. When we can we will use Greek letters for meta-variables, as this makes it impossible to confuse them with LISP variables. But in any case, we will always give a **range** for any meta-variable, using '$\in$' or the words 'is a' to define a set in which the values of the meta-variable must lie, as in '$\alpha \in$ integer' or '`x is an integer`'.

When we give rewrite rules in code, we will generally use the form:

```
meta-expression-1 ===> meta-expression-2
                              conditions
ranges
```

where the 'ranges' name the meta-variables and restrict each to a set that is its range, and the **conditions** give special conditions that must be true for the meta-variable values in the given meta-rule. An example is:

```
;; (factorial n) ===> (* n (factorial (- n 1)))
;;                         if (> n 0)
;; where n is an integer
```

Here the condition is '`if (> n 0)`', the range is '`where n is an integer`', and the meta-variable is '`n`'.

Exercise 1.4.6.1 Use rewrite rules to define the following functions (do <u>not</u> write any `defun`'s):

(a) A function that computes the hypotenuse of a right triangle given the sides. (In COMMONLISP the `sqrt` function computes square roots.)
(b) The `oddp` function of Exercise 1.4.4.1.
(c) The function of Exercise 1.4.5.1.
(d) The function of Exercise 1.4.5.2.

1.4.7 Call Trees

Let us evaluate `(factorial 2)` by using rewrite rules. To do so we will need a few rewrite rules we have not yet introduced, plus the concept of a function call.

The new rewrite rules are to be used in evaluating `cond` expressions one step at a time:

$$\begin{array}{l}\texttt{(cond (t } v_1\texttt{)} \\ \qquad\quad \texttt{(}e_2\ v_2\texttt{)} \\ \qquad\quad \texttt{...)}\end{array} \Longrightarrow v_1$$

$$\begin{array}{l}\texttt{(cond (nil } v_1\texttt{)} \\ \qquad\quad \texttt{(}e_2\ v_2\texttt{)} \\ \qquad\quad \texttt{...)}\end{array} \Longrightarrow \begin{array}{l}\texttt{(cond (}e_2\ v_2\texttt{)} \\ \qquad\quad \texttt{...)}\end{array}$$

Also, the rewrite rule we will use for evaluating the factorial function itself is derived from the LISP function definition of `factorial`, and is:

```
(factorial n)  ==>  (cond ((= n 0) 1)
                          ((> n 0) (* n (factorial (- n 1)))))
```

where `n` is an integer, `n`≥`0`

With the help of these rules we may write:

```
(factorial 2)  ==>  (cond ((= 2 0) 1)
                          ((> 2 0) (* 2 (factorial (- 2 1)))))
               ==>  (cond (nil     1)
                          ((> 2 0) (* 2 (factorial (- 2 1)))))
               ==>  (cond ((> 2 0) (* 2 (factorial (- 2 1)))))
               ==>  (cond (t (* 2 (factorial (- 2 1)))))
               ==>  (* 2 (factorial (- 2 1)))
               ==>  (* 2 (factorial 1)) ==> (* 2 1) ==> 2
                                 ⇓ call
(factorial 1)  ==>  (cond ((= 1 0) 1)
                          ((> 1 0) (* 1 (factorial (- 1 1)))))
               ==>  (cond (nil     1)
                          ((> 1 0) (* 1 (factorial (- 1 1)))))
               ==>  (cond ((> 1 0) (* 1 (factorial (- 1 1)))))
               ==>  (cond (t (* 1 (factorial (- 1 1)))))
               ==>  (* 1 (factorial (- 1 1)))
               ==>  (* 1 (factorial 0)) ==> (* 1 1) ==> 1
                                 ⇓ call
(factorial 0)  ==>  (cond ((= 0 0) 1)
                          ((> 0 0) (* 0 (factorial (- 0 1)))))
               ==>  (cond (t       1)
                          ((> 0 0) (* 0 (factorial (- 0 1)))))
               ==>  1
```

Here we have written the evaluation of `(factorial 2)` out using rewrite rules, except that when we had to evaluate `(factorial 1)`, we made a '**call**' on a separate and distinct computation to find out what `(factorial 1)` rewrote to. This is rather like making a telephone call to get an answer, which is why it is called making a 'call'.

While the answer requested is being computed, the call is in progress, and when the answer has been computed, the call ends and **returns** the answer.

During the first call to evaluate `(factorial 1)` we made a second call to evaluate `(factorial 0)`. This second call is made during the first call and ends before the first call ends.

Fortunately there was no need to make any calls to evaluate `(factorial 0)`.

The computations of `(factorial 2)`, `(factorial 1)`, and `(factorial 0)` above are referred to as '**function executions**'. Function executions can call each other. More specifically, a first function execution can place a call to start a second function execution, and when the second function execution finishes, the call terminates and a value is returned to the first function execution.

Each call is made 'to' a function and provides **arguments** for that function. The first call above is to the `factorial` function and has the single argument `1`. The second call is also to `factorial` but with argument `0`.

These calls are said to be '**recursive calls**' because the function `factorial` 'is calling itself'. The evaluation of `(factorial 0)` is called the '**base case**' of the recursion because it makes no recursive calls.

An abbreviation for the evaluation above is:

$$\begin{array}{c} \texttt{(factorial 2)} \Longrightarrow 2 \\ \Downarrow \\ \texttt{(factorial 1)} \Longrightarrow 1 \\ \Downarrow \\ \texttt{(factorial 0)} \Longrightarrow 1 \end{array}$$

This is essentially a labeled graph, with nodes that represent function executions, and arrows that represent calls between function executions. This graph is actually a tree, and is referred to as the '**call tree**' of the evaluation of `(factorial 2)`.

In general the nodes of a call tree are labeled by:

$$\text{expression to be evaluated} \Longrightarrow \text{value to be returned}$$

Sometimes only the expression to be evaluated, or even only the name of the function, is used as the node label.

The arrows in a call tree may be labeled with the integers `1`, `2`, `3`, ... that indicate the order in which the calls were made during the execution of a function.

Figure 1.7 gives three example call trees.

The COMMONLISP interpreter has a '**trace**' mode that displays the data contained in a call tree in a different, less pictorial, form. The call tree for `factorial` in Figure 1.7 corresponds to the trace in Figure 1.6.

To activate the trace mode for the `factorial` function, the interpreter is asked to evaluate the expression:

```
(trace factorial)
```

as is done in Figure 1.6. Functions for which trace mode is active are called '**traced functions**'.

Whenever an execution of a traced function begins, a line is printed containing the word '`Trace`' and the expression to be evaluated by the function execution. When the function execution ends, a line is printed containing '`Trace`', the name of the function, the symbol '`==>`', and lastly the value returned by the function execution.

Each of these lines has a number which corresponds to the level of the function execution as a node in the call tree. The root at the top of the call tree is level 1. The nodes just below the root are level 2, and the nodes just below these are level 3, etc.

The other call trees in Figure 1.7 correspond to the code and traces given in Figures 1.8 and 1.9. Figure 1.8 defines the `power` function that computes integer powers of any number. This function has the same basic structure as the `factorial` function, and generates similar call trees.

Figure 1.9 defines the `fast-power` function that uses the tricks illustrated by the example computation:

$$2^6 \Longrightarrow (2^3)^2 \Longrightarrow (2 \cdot (2^2))^2$$

to compute powers much more rapidly than the `power` function can. The `fast-power` function also calls the `square` function defined in Figure 1.3.

We will discuss the relative speed of `power` and `fast-power` in more detail in the next section. One can see from Figure 1.7 that when the input argument doubles the

$$\begin{tikzcd}
\texttt{(factorial 5)} \Longrightarrow 120 \arrow[d, "1"] \\
\texttt{(factorial 4)} \Longrightarrow 24 \arrow[d, "1"] \\
\texttt{(factorial 3)} \Longrightarrow 6 \arrow[d, "1"] \\
\texttt{(factorial 2)} \Longrightarrow 2 \arrow[d, "1"] \\
\texttt{(factorial 1)} \Longrightarrow 1 \arrow[d, "1"] \\
\texttt{(factorial 0)} \Longrightarrow 1
\end{tikzcd}
\qquad
\begin{tikzcd}
\texttt{(power 2 6)} \Longrightarrow 64 \arrow[d, "1"] \\
\texttt{(power 2 5)} \Longrightarrow 32 \arrow[d, "1"] \\
\texttt{(power 2 4)} \Longrightarrow 16 \arrow[d, "1"] \\
\texttt{(power 2 3)} \Longrightarrow 8 \arrow[d, "1"] \\
\texttt{(power 2 2)} \Longrightarrow 4 \arrow[d, "1"] \\
\texttt{(power 2 1)} \Longrightarrow 2 \arrow[d, "1"] \\
\texttt{(power 2 0)} \Longrightarrow 1
\end{tikzcd}$$

$$\begin{tikzcd}
 & & & \texttt{(fast-power 2 6)} \Longrightarrow 64 \arrow[dl, "1"'] \arrow[dr, "2"] & \\
 & & \texttt{(fast-power 2 3)} \Longrightarrow 8 \arrow[dl, "1"'] \arrow[dr, "2"] & & \texttt{(square 8)} \Longrightarrow 64 \\
 & \texttt{(fast-power 2 1)} \Longrightarrow 2 \arrow[dl, "1"'] \arrow[dr, "2"] & & \texttt{(square 2)} \Longrightarrow 4 & \\
\texttt{(fast-power 2 0)} \Longrightarrow 1 & & \texttt{(square 1)} \Longrightarrow 1 & &
\end{tikzcd}$$

Figure 1.7: Example Call Trees

```
;; (power m 0) ===> 1
;; (power m n) ===> (* m (power m (- n 1)))
;;                  if (> n 0)
;;
;;    where m is a number and n is an integer
;;
(defun power (m n)
  (cond ((= n 0) 1)
        ((> n 0)
         (* m (power m (- n 1))))))
--->
POWER

(power 3 2)
--->
9

(trace power)
--->
;; Tracing function POWER.
(POWER)

(power 2 6)
--->

1. Trace: (POWER '2 '6)
2. Trace: (POWER '2 '5)
3. Trace: (POWER '2 '4)
4. Trace: (POWER '2 '3)
5. Trace: (POWER '2 '2)
6. Trace: (POWER '2 '1)
7. Trace: (POWER '2 '0)
7. Trace: POWER ==> 1
6. Trace: POWER ==> 2
5. Trace: POWER ==> 4
4. Trace: POWER ==> 8
3. Trace: POWER ==> 16
2. Trace: POWER ==> 32
1. Trace: POWER ==> 64
64
```

Figure 1.8: Unoptimized Power Function

```
;; (fast-power m 0) ===> 1
;; (fast-power m n)
;;     ===> (square (fast-power m (/ n 2)))
;;          if (evenp n) and (/= n 0)
;;     ===> (* m (square (fast-power m (/ (- n 1) 2)))
;;          if (oddp n)
;;
;;    where m is a number and n is an integer
;;
(defun fast-power (m n)
  (cond ((= n 0) 1)
        ((evenp n)
              (square (fast-power m (/    n    2))))
        ((oddp n)
         (* m (square (fast-power m (/ (- n 1) 2)))))))
--->
FAST-POWER

(fast-power 3 2)
--->
9

(trace fast-power square)
--->
;; Tracing function FAST-POWER.
;; Tracing function SQUARE.
(FAST-POWER SQUARE)

(fast-power 2 6)
--->

1. Trace: (FAST-POWER '2 '6)
2. Trace: (FAST-POWER '2 '3)
3. Trace: (FAST-POWER '2 '1)
4. Trace: (FAST-POWER '2 '0)
4. Trace: FAST-POWER ==> 1
4. Trace: (SQUARE '1)
4. Trace: SQUARE ==> 1
3. Trace: FAST-POWER ==> 2
3. Trace: (SQUARE '2)
3. Trace: SQUARE ==> 4
2. Trace: FAST-POWER ==> 8
2. Trace: (SQUARE '8)
2. Trace: SQUARE ==> 64
1. Trace: FAST-POWER ==> 64
64
```

Figure 1.9: Optimized Power Function

call tree of '`power`' nearly doubles in size, but the call tree of '`fast-power`' just adds 2 nodes. As the execution time of a function call tends to be proportional to the size of the call tree rooted at the function call, there is quite a difference in the execution time of '`power`' and '`fast-power`'. For example, the call tree of '`(power 1024)`' has 1025 nodes while the call tree of '`(fast-power 1024)`' has only 23 nodes.

Call trees have at least four uses. First, they can be used to understand and to demonstrate the understanding of algorithms. Second, size estimations of call trees can be used to estimate the execution times of algorithms. Third, the call tree is itself a data structure which may substitute for a data structure needed by the algorithm. For example, in Chapter 3 we encounter search algorithms that search a tree which they need to partly keep track of. For the particular case of depth-first search, the call tree of the algorithm mirrors this search tree, and the call tree by itself can be used to keep track of the search tree. Fourth and lastly, any mismatch between the actual call tree of an execution and the call tree as the programmer thinks it should be can be used to debug code. The call tree of an actual execution can be discovered by tracing the execution, as is done, for example, in Figure 1.9.

If you indent each trace line by a number of spaces equal to the twice the level number printed on the line, and if you then connect with vertical bars the level numbers on the pair of lines that begin and end each function execution, you have a pictorial representation of the call tree turned on its side, with the root at level 1 on the left. For example, the trace from Figure 1.9 appears as:

```
1. Trace: (FAST-POWER '2 '6)
| 2. Trace: (FAST-POWER '2 '3)
| | 3. Trace: (FAST-POWER '2 '1)
| | | 4. Trace: (FAST-POWER '2 '0)
| | | 4. Trace: FAST-POWER ==> 1
| | | 4. Trace: (SQUARE '1)
| | | 4. Trace: SQUARE ==> 1
| | 3. Trace: FAST-POWER ==> 2
| | 3. Trace: (SQUARE '2)
| | 3. Trace: SQUARE ==> 4
| 2. Trace: FAST-POWER ==> 8
| 2. Trace: (SQUARE '8)
| 2. Trace: SQUARE ==> 64
1. Trace: FAST-POWER ==> 64
```

Some LISP interpreters print traces in this format. In any case, it is a fairly mindless exercise to turn a trace into a picture of a call tree.

One reason this is true is that at any time at most one function execution can exist on each level N, and any function execution that exists on level $N+1$ must have been called by the currently executing function execution on level N.

An interesting question is why evaluations of expressions such as `(* 2 1)` during the execution of `(factorial 2)` are not considered to be function executions for the purposes of displaying the call tree. In a sense, this is just a matter of an 'arbitrary choice'. Call trees can include or omit any function executions you like.

However, there is another reason. LISP interpreters do a fair amount of work to set up a call to a function defined by '`defun`', and the functions in our call trees are just those defined by '`defun`'. Other functions, such as `*`, are built into the LISP interpreter, and execute much more quickly. Therefore it is usual to exclude such '**builtin functions**' from call trees.

Notice that in Figure 1.9 tracing is activated for both the `fast-power` and `square` functions. Choosing the functions for which to activate trace is analogous to choosing the functions that will appear in a call tree.

Exercise 1.4.7.1 Draw the call tree for the execution of the following. Label nodes with the called function name, the arguments, and the returned value.

(a) `(f 3)` where `f` is the **Fibonacci function**[15]defined by the rewrite rules:

```
(f 0) ===> 0
(f 1) ===> 1
(f n) ===> (+ (f (- n 1)) (f (- n 2)))
                if n > 2
where n is an integer
```

(b) `(c 6 3)` where `c` is the function defined in Exercise 1.4.5.1.[16]

[16]The literature does not agree about where the Fibonacci function starts. Consider three definitions of the Fibonacci function, `f`, `f1`, and `f2`, all with the same recursive definition for `n` ≥ 2, but with `(f 0)===>0`, `(f 1)===>1`, `(f1 0)===>1`, `(f1 1)===>1`, `(f2 0)===>1`, `(f2 1)===>2`. Then (`f1 n`) = (`f` (+ `n 1`)), (`f2 n`) = (`f1` (+ `n 1`)) = (`f` (+ `n 2`)). [Rob84] defines the Fibonacci function to be `f1`, and others use `f2`. We use `f` because historically Computer Science 51 at Harvard has used `f`.

[16]The function `c` is the **binomial coefficient function**, and the call trees of `c` are pieces of the Pascal triangle with duplicated nodes: see [Rob84].

1.4.8 Complexity Theory

The behavior of the execution time of a program or function as the size of its input increases is called the **time complexity** of the program.

The execution time of a function execution may be proportional to the size of its call tree, that is, to the number of nodes in the call tree. Careful inspection of Figure 1.7 reveals that:

call tree size of `(factorial n)`	=	n+1
call tree size of `(power m n)`	=	n+1
call tree size of `(fast-power m n)`	=	$2 \cdot \lfloor \log_2 \mathtt{n} \rfloor + 3$ for $n > 0$

where $\lfloor \log_2 \mathtt{n} \rfloor$ is the greatest integer not larger than the base 2 logarithm of `n`, and a **call tree size** is the number of nodes in the tree.

Complexity is measured using the '**big O**' notation. Following the lead of mathematicians, we define:

$$= O(F(n)) \quad \text{means} \quad \leq K_1 \cdot F(n) \text{ for all } n > K_2, \text{ for some constants } K_1,\ K_2$$

Here n is the measure of the size of the input, and $F(n)$ is some function of n, such as $F(n) = n + 1$ or $F(n) = \log n$. With this notation we will show that:

execution time of `(factorial n)`	=	$O(\mathtt{n})$
execution time of `(power m n)`	=	$O(\mathtt{n})$
execution time of `(fast-power m n)`	=	$O(\log \mathtt{n})$

if the execution times are proportional to the number of nodes in the call trees.

It turns out that[17]

[17] If we are given C_1 and C_2, take K_2 large enough so that $F(n)$ is defined and non-decreasing as n increases for $n \geq K_2$, and then take $K_1 = C_2 + \frac{C_1}{F(K_2)}$. Note that when $F(n) = \frac{1}{n}$, which decreases with increasing n, we have $1 \leq 1 + 1 \cdot \frac{1}{n}$ but $1 \neq O(\frac{1}{n})$.

Conversely, if we are given K_1 and K_2, take $C_2 = K_1$, and take C_1 to be the maximum of all defined values to the left of the '$\leq C_1 + C_2 \cdot F(n)$' for $n \leq K_2$.

$$\begin{array}{lll} = O(F(n)) & \text{means} & \leq C_1 + C_2 \cdot F(n) \text{ whenever } F(n) \text{ is defined,} \\ & & \text{for some constants } C_1,\, C_2, \\ & \text{provided} & F(n) \text{ is defined and non-decreasing as } n \text{ increases} \\ & & \text{for sufficiently large } n \end{array}$$

In computer complexity theory, $F(n)$ is always defined and non-decreasing for sufficiently large n, and this last criterion for being $O(F(n))$ is easier to apply than the raw definition.

In particular, we can demonstrate that the execution time of `(fast-power m n)` $=$ $O(\log \mathtt{n})$ by taking $C_1 = 2 \cdot \log_2 10 \cdot C_e$ and $C_2 = 3 \cdot C_e$, where C_e is the maximum execution time of a single function execution, since $\log_2 \mathtt{n} = (\log_2 10) \cdot (\log\ \mathtt{n})$.

One good question is: do all the function executions in the call tree have a maximum execution time C_e. For modern computers and the functions we are considering here the answer is yes if $\mathtt{n} \leq 10^9$ and either `m` is an integer with $\mathtt{m} \leq 10^9$ or `m` is floating point. When an integer becomes larger than 10^9, the execution time of addition grows proportionately to the size of the integer in bits, and the execution time of multiplication grows proportionately to the product of the sizes of the operands in bits. This introduces a lot of complexity in the results, which, in case the reader is curious, become:

$$\begin{array}{lcl} \text{execution time of } \mathtt{(factorial\ n)} & = & O((\mathtt{n} \cdot \log\ \mathtt{n})^2) \\ \text{execution time of } \mathtt{(power\ m\ n)} & = & O((\mathtt{n} \cdot \log\ \mathtt{m})^2) \\ \text{execution time of } \mathtt{(fast\text{-}power\ m\ n)} & = & O((\mathtt{n} \cdot \log\ \mathtt{m})^2) \end{array}$$

if `m` is an integer, and

$$\begin{array}{lcl} \text{execution time of } \mathtt{(power\ m\ n)} & = & O(\mathtt{n} \cdot \log\ \mathtt{n}) \\ \text{execution time of } \mathtt{(fast\text{-}power\ m\ n)} & = & O((\log\ \mathtt{n})^2) \end{array}$$

if `m` is floating point.

Complexity theory expresses the execution time of a program or function using the big O notation. This gives an abstraction of the true execution time of the program.

If we say that a program or function has **time complexity** $O(F(n))$, we mean that the execution time of the program or function is $O(F(n))$ where n measures the size of the input. If we say that an **algorithm** has time complexity $O(F(n))$, we mean any program or function that just implements the algorithm has that time complexity.

Suppose we are told the time complexity of a particular program is $O(n)$, and we are asked about the ratio between the execution time of the program on input of N items and on input of $1000 \cdot N$ items. What do we know? We know that we have an estimate of the maximum execution time of the program, and that estimate for an input of $1000 \cdot N$ items is

$$\frac{C_1 \cdot 1000 \cdot N + C_2}{C_1 \cdot N + C_2} \leq 1000$$

times larger than the estimate for N items.

If instead we had been told the execution time was $O(\log\ n)$, then we would know the estimate for an input of $1000 \cdot N$ items is only

$$\frac{C_1 \cdot (3 + \log\ N) + C_2}{C_1 \cdot \log\ N + C_2} \leq \frac{3 + \log\ N}{\log\ N} \leq 3$$

times larger than the estimate for N items.

This is all we know, which is the price of dealing with abstractions.

Of course complexity theory also deals with estimates of minimum execution times, in case you are suspicious that the actual execution times are much less than the maximum estimates. However, we will not get into minimum time estimates in this book, although it is true that the time estimates we gave above can be made into minimum estimates by just changing the constants.

An algorithm that is $O(1)$ executes in constant time that is independent of the size of the input. If $0 \leq \alpha < \beta$, then in general, an algorithm that is $O(n^\alpha)$ is likely to be faster for large input than an algorithm that is $O(n^\beta)$, by a factor that approaches $n^{(\beta-\alpha)}$ as the input size increases. An algorithm that is $O(n^\alpha \cdot \log\ n)$, on the other hand, may be only a small factor slower than an algorithm that is $O(n^\alpha)$ for any realistic input size.

Exercise 1.4.8.1 Show that the size of the call tree of `(c n m)` $= O(2^n)$, where `c` is the function defined in Exercise 1.4.5.1.[18]

Exercise 1.4.8.2 [Hard] Show the execution time of `(factorial n)` $= O((\mathtt{n}\cdot \log\ \mathtt{n})^2)$ with addition and multiplication times as described in this section for arbitrarily large `n`.

[18] Of course if one writes a non-recursive `c` that calls the recursive `factorial` function, the call tree needed to compute `(c n m)` has size $= O(n)$.

Exercise 1.4.8.3 [Hard] Find the smallest real number $\alpha > 0$ such that the size of the call tree of `(f n)` $= O(\alpha^n)$, where `f` is the Fibonnaci function defined in Exercise 1.4.7.1.[19]

1.5 LISP Symbolic Data

In this section we describe how symbolic data is stored in the memory of a LISP system, and what elementary computations LISP can perform with symbolic data.

1.5.1 Symbolic Expressions (S-Expressions)

LISP stores symbolic data as '**symbolic expressions**', also called '**s-expressions**'. One possible representation for s-expressions is:

s-expression	::=	atom | list
atom	::=	number | symbol
list	::=	(s-expression*)

Here ::= means 'is', | means 'or', a blank space separating two items means 'is followed by', and * means 'repeated 0 or more times'. Thus the above is shorthand for:

an s-expression	is	an atom or a list
an atom	is	a number or a symbol
a list	is	a `(`, followed by an s-expression repeated 0 or more times, followed by a `)`

The representation we have just given to define an 's-expression' is called a **grammar**, and consists of three grammar equations, each containing one '::='.

A **symbol** can be a sequence of letters, digits, and any of the following characters:

```
+ - * / = < > ~ @ $ % ^ & _ . ? ! [ ] { }
```

[19]There is a clever way to write a recursive `f` that computes the Fibonnaci function values of increasing integers and has a call tree of size $= O(n)$.

However, any sequence of such characters that is a number is not a symbol. Also, as we will see shortly, the one-character sequence '`.`' is not a symbol.

Some example symbols are:

```
+      -      *       /      circle-area
<      <=     >       >=     /=
=      floor  mod     oddp   evenp
nil    t      cond    defun  factorial
trace  max    square  power  fast-power
m      n      m2      2m     my-max
```

When a LISP interpreter reads a symbol, it immediately converts lower case letters to upper case, so when the symbol is printed it prints with all upper case letters, and when two symbols are compared for equality, all letters in the symbols are treated as upper case. For example, '`nil`', '`Nil`', and '`NIL`' are all equivalent when typed into a LISP interpreter.

Later we will introduce symbols that contain escaped characters, including possibly escaped lower case letters that are not converted to upper case.

The above way of representing s-expressions does not make it easy for computers to store expressions, because lists can have varying lengths. A representation that is better for computers is:

s-expression	::=	atom $\mid$ cons-cell
atom	::=	number $\mid$ symbol
cons-cell	::=	(s-expression . s-expression)
non-empty-list	::=	(first-element . rest-of-list)
empty-list	::=	`nil`

where `nil` $\in$ symbol

The main idea here is that a non-empty list will be represented by a '**cons-cell**', which is a data structure with exactly two parts. The first part represents the first element of the list, and the second part represents the rest of the list. Because there are always exactly two parts, the cons-cell is easier for a computer to handle than a variable length list.

The empty list must be represented in some other way. In LISP the empty list is represented by the '`nil`' symbol.

On input, a cons-cell can be represented by the representatives of its two parts, separated by a '`.`', and surrounded by parentheses.

Then lists such as '`(+ 2 3)`' must be translated into cons-cells plus uses of '`nil`' to represent the empty list. This is done when the lists are input with the help of the equations:

$$
\begin{array}{lcl}
\texttt{()} & \equiv & \texttt{nil} \\
\texttt{(}\ \alpha\ \texttt{)} & \equiv & \texttt{(}\ \alpha\ \texttt{.}\ \texttt{nil}\ \texttt{)} \\
\texttt{(}\ \alpha\ \beta \ldots \texttt{)} & \equiv & \texttt{(}\ \alpha\ \texttt{.}\ \texttt{(}\ \beta \ldots \texttt{))}
\end{array}
$$

$$\text{where } \alpha \in \text{s-expression}, \quad \beta \in \text{s-expression}$$

Here and throughout this chapter, $\equiv$ means 'is equivalent when input'. Some examples that apply the equations just given are:

$$
\begin{array}{lclcl}
\texttt{()} & \equiv & \texttt{nil} & & \\
\texttt{(1)} & \equiv & \texttt{(1 . nil)} & & \\
\texttt{(1 2)} & \equiv & \texttt{(1 . (2))} & \equiv & \texttt{(1 . (2 . nil))} \\
\texttt{(1 2 3)} & \equiv & \texttt{(1 . (2 3))} & \equiv & \texttt{(1 . (2 . (3)))} \\
 & & & \equiv & \texttt{(1 . (2 . (3 . nil)))} \\
\texttt{(+ 2 3)} & \equiv & \texttt{(+ . (2 3))} & \equiv & \texttt{(+ . (2 . (3)))} \\
 & & & \equiv & \texttt{(+ . (2 . (3 . nil)))}
\end{array}
$$

Note that usually notation we introduce is used throughout computer science, but our use of $\equiv$ to mean 'is equivalent when input' is an exception, and is only used by us in this chapter.

There is a problem inputting symbolic data such as '`(+ 5 6)`' to a LISP interpreter. The interpreter will try to evaluate such data. To protect it from evaluation, it must be **quoted**, which in in LISP is done by placing the single character `'` in front of the datum. Thus:

```
(+ 2 3)
--->
5

'(+ 2 3)
--->
(+ 2 3)
```

The single quote obeys the rule:

$$\texttt{'}\alpha \equiv \texttt{(quote }\alpha\texttt{)}$$

$$\text{where } \alpha \in \text{s-expression}$$

Thus `'` is just a new method of input, but not a new way of storing data. The '`quote`' function takes one unevaluated argument and returns its value. The quote function is unusual in that its argument is not evaluated before the `quote` function evaluates. In LISP such functions are called '**special functions**'. Other special functions we have seen are `cond` and `defun`.

A number of examples of s-expression input and use of quoted expressions are given in Figure 1.10.

Note the general principal that when printing an s-expression, the LISP interpreter tries to choose the most compact of the various input-equivalent representations of the s-expression. Thus expressions using the '`quote`' function are printed with the `'` character.

Note that it is necessary to quote a symbol if you want to input it as data. Otherwise the LISP interpreter will treat the symbol as a variable name and try to find the value of the named variable.

However, one need not quote numbers or the symbols `nil` and `t`. These evaluate to themselves.

We have introduced two closely related notations, $\Longrightarrow$ meaning 'rewrites to', and `--->` meaning 'evaluates to'. We can now relate these notations precisely:

$$\alpha \texttt{ ---> } \beta \qquad \text{means} \qquad \alpha \Longrightarrow \texttt{'}\beta$$

$$\text{where } \alpha,\, \beta \in \text{s-expression}$$

Any list of two or more s-expressions in parentheses with a '`.`' just before the last s-expression represents a list in which the last s-expression represents the 'rest of the list'. If this last s-expression is an atom other than `nil`, we have an anomalous datum that is not a true list. Such a datum is called a **dotted list.** Examples of a dotted and an undotted list are:

$$\texttt{(a b c . d)} \equiv \texttt{(a . (b c . d))} \equiv \texttt{(a . (b . (c . d)))}$$

$$\texttt{(a b c)} \equiv \texttt{(a b c . nil)} \equiv \texttt{(a . (b c . nil))} \equiv \texttt{(a . (b . (c . nil)))}$$

```
;; Computer Representation of Lists

'(a b)
--->
(A B)

'(a . (b))
--->
(A B)

'(a . (b . ()))
--->
(A B)

'(a . (b . nil))
--->
(A B)

'(5 . ((a . (b . nil)) .  (7 . nil)))
--->
(5 (A B) 7)

'(a . (b . c))
--->
(A B . C)

;; Use of Quote (')

(* 2 5)
--->
10

'(* 2 5)
--->
(* 2 5)

;; `m' without a quote gives an error
;;
m
--->
ERROR: EVAL: variable M has no value
NIL

'm
--->
M
```

```
;; More Quotes

'Hi-ThErE
--->
HI-THERE

'5
--->
5

5
--->
5

'nil
--->
NIL

nil
--->
NIL

't
--->
T

t
--->
T

''x
--->
'X

'(quote x)
--->
'X

''(a 'b c)
--->
'(A 'B C)

'(quote (a (quote b) c))
--->
'(A 'B C)
```

Figure 1.10: Quoted Expression Examples

Exercise 1.5.1.1 Which of the following s-expressions are equivalent when input ($\equiv$):

(a)	`(a . (b))`	(g)	`(a () b)`
(b)	`(a nil b)`	(h)	`((a ()) b)`
(c)	`(a . (b . nil))`	(i)	`(a b . nil)`
(d)	`(a . b)`	(j)	`((a . nil) b)`
(e)	`(a . (b . ()))`	(k)	`(a b)`
(f)	`((a) . (b))`	(l)	`((a nil) b)`

Exercise 1.5.1.2 Which of the following s-expressions are equivalent when input ($\equiv$):

(a)	`(a 'b)`	(g)	`(a . 'b)`
(b)	`(a '(b))`	(h)	`(a (quote b))`
(c)	`(a quote . (b))`	(i)	`(a quote (b))`
(d)	`(a (quote (b)))`	(j)	`(a quote b)`
(e)	`(a quote . b)`	(k)	`(a . (quote . b))`
(f)	`(a (quote . ((b)))`	(l)	`(a . ((quote b)))`

1.5.2 Manipulation of List Data

The three lowest level functions that manipulate cons-cells are the `cons` function that creates a new cons-cell, the `car` function that returns the first of the two parts of the cons-cell, and the `cdr` function that returns the second of the two parts of the cons-cell. These functions can be defined using the rewrite rules:

$$
\begin{array}{lcl}
\texttt{(cons '}\alpha\texttt{ '}\beta\texttt{)} & \Longrightarrow & \texttt{'(}\alpha\texttt{ . }\beta\texttt{)} \\
\texttt{(car '(}\alpha\texttt{ . }\beta\texttt{))} & \Longrightarrow & \texttt{'}\alpha \\
\texttt{(car '())} & \Longrightarrow & \texttt{nil} \\
\texttt{(cdr '(}\alpha\texttt{ . }\beta\texttt{))} & \Longrightarrow & \texttt{'}\beta \\
\texttt{(cdr '())} & \Longrightarrow & \texttt{nil}
\end{array}
$$

where $\alpha \in$ s-expression, $\beta \in$ s-expression

Note that these are rewrite meta-rules, which represent an infinite number of rewrite rules including:

`(cons 'b '5)`	$\Longrightarrow$	`'(b . 5)`
`(car '(b . 5))`	$\Longrightarrow$	`'b`
`(cdr '(b . 5))`	$\Longrightarrow$	`'5`
`(car '((a b c) . 5))`	$\Longrightarrow$	`'(a b c)`
`(cdr '((a b c) . 5))`	$\Longrightarrow$	`'5`
`(car '(5 . (a b c))`	$\Longrightarrow$	`'5`
`(cdr '(5 . (a b c))`	$\Longrightarrow$	`'(a b c)`

Some other examples that require replacing expressions by their input equivalents are:

`(cons 'a '(b c))`	$\Longrightarrow$	`'(a . (b c))`	$\equiv$	`'(a b c)`
`(car '(a b c))`	$\equiv$	`(car '(a . (b c)))`	$\Longrightarrow$	`'a`
`(cdr '(a b c))`	$\equiv$	`(cdr '(a . (b c)))`	$\Longrightarrow$	`'(b c)`
`(cons 'a nil)`	$\Longrightarrow$	`'(a . nil)`	$\equiv$	`'(a)`

The names `car` and `cdr` have historic origins.[20] In COMMONLISP the function `first` is defined to do the same thing as the function `car`, namely to return the first element of a non-empty list; and the function `rest` is defined to do the same thing as the function `cdr`, namely to return the rest of a non-empty list.

Examples of list manipulation functions interacting with the LISP interpreter are given in Figure 1.11.

Examples of higher level functions that manipulate lists are given in Figures 1.12, 1.13, and 1.14. These examples show how the COMMONLISP `nth`, `append`, and `reverse` functions might be coded. In order to avoid conflict with the actual COMMONLISP functions, the functions in the examples are given the names `my-nth`, `my-append`, and `my-reverse`.

The `my-append` and `my-reverse` functions make use of the COMMONLISP `null` function that returns true if its operand is the empty list `nil`, and returns false otherwise.

The `my-reverse` function also makes use of an `&optional` argument. In

[20]The memory words of the old 7090 computer held two numbers, both large enough to be addresses, that were called the 'address' and the 'decrement'. These numbers held the two pointers in the cons cell, which was a single word on a 7090. `car` was the name of the instruction that read the the 'address' part of the memory word, and `car` stands for 'contents of address register'. `cdr` read the 'decrement' and stands for 'contents of decrement register'.

`(cons 'a '(b c))`
`--->`
`(A B C)`
	`(cons 'a nil)`
	`--->`
	`(A)`

`(car '(a b c))`
`--->`
`A`
	`(cdr '(a b c))`
	`--->`
	`(B C)`

`(first '(a b c))`
`--->`
`A`
	`(rest '(a b c))`
	`--->`
	`(B C)`

Figure 1.11: Functions That Manipulate List Data

```
(defun my-reverse (w &optional (z nil))
```

the '`&optional`' means that all arguments afterwards are **optional arguments**. The value of an optional argument may or may not be given when the function is called. If not given, the argument is assigned a **default value**.

In the case of `my-reserve`, there is one optional argument named '`z`' whose default value is '`nil`'. In the call `(my-reverse '(a) '(b))` the value of `'(b)` is assigned to `z`, but in the call `(my-reverse '(a))` the default value of `nil` is assigned to `z`.

In general, each argument after '`&optional`' is described by a two element list giving first the variable name and second the default value for the optional argument.

When writing recursive functions, optional arguments are often used to represent values needed by the recursive steps that are not initially input to the recursion. In the case of `my-reverse`, the value of `z` is the output to be returned, i.e., the reversed list.

In COMMONLISP there are 28 functions that abbreviate various combinations of `car` and `cdr`, namely:

```
;; (my-nth 0 'x) ===> (car 'x)
;; (my-nth n 'x) ===> (my-nth (- n 1) (cdr 'x))
;;                    if n > 0
;;
;; where n is an integer,  x is an s-expression
;;
(defun my-nth (n x)
  (cond
   ((= n 0) (car x))
   ((> n 0) (my-nth (- n 1) (cdr x)))))
--->
MY-NTH

(my-nth 0 '(a b c))
--->
A

(my-nth 1 '(a b c))
--->
B

(my-nth 2 '(a b c))
--->
C

(my-nth 3 '(a b c))
--->
NIL

;; (my-nth -1 '(a b c)) would loop `forever'

;; The following is in error:
;;
(my-nth 'a '(a b c))
--->
ERROR: argument to = should be a number: A
NIL
```

Figure 1.12: The `Nth` Function

```
;; (my-append '() 'z) ===> 'z
;; (my-append '(x . y) 'z) ===>
;;      (cons 'x (my-append 'y 'z))
;;
;; where x, y, z are s-expressions
;;
(defun my-append (w z)
  (cond
   ((null w) z)
   (t (cons (car w) (my-append (cdr w) z)))))
--->
MY-APPEND

(my-append '() '(1 2))
--->
(1 2)

(my-append '(a) '(1 2))
--->
(A 1 2)

(my-append '(a b) '(1 2))
--->
(A B 1 2)

(my-append '(a b) '(1))
--->
(A B 1)

(my-append '(a b) '())
--->
(A B)
```

Figure 1.13: The `Append` Function

```
;; (my-reverse 'x)  ===> (my-reverse 'x '())
;; (my-reverse '() 'z) ===> 'z
;; (my-reverse '(x . y) 'z) ===>
;;      (my-reverse 'y '(x . z))
;;
;; where x, y, z are s-expressions
;;
(defun my-reverse (w &optional (z nil))
  (cond
   ((null w) z)
   (t (my-reverse (cdr w) (cons (car w) z)))))
--->
MY-REVERSE

(my-reverse '(a b) '(1 2))
--->
(B A 1 2)

(my-reverse '(a b c d) nil)
--->
(D C B A)

(my-reverse '(a b c d))
--->
(D C B A)
```

Figure 1.14: The `Reverse` Function

`(caar '`α`)`	$\Longrightarrow$	`(car (car '`α`))`	where $\alpha, \in$ s-expression
`(cadr '`α`)`	$\Longrightarrow$	`(car (cdr '`α`))`	
`(cdar '`α`)`	$\Longrightarrow$	`(cdr (car '`α`))`	
`(cddr '`α`)`	$\Longrightarrow$	`(cdr (cdr '`α`))`	
`(caaar '`α`)`	$\Longrightarrow$	`(car (car (car '`α`)))`	
`(caadr '`α`)`	$\Longrightarrow$	`(car (car (cdr '`α`)))`	
`(cadar '`α`)`	$\Longrightarrow$	`(car (cdr (car '`α`)))`	
....			
`(cdddar '`α`)`	$\Longrightarrow$	`(cdr (cdr (cdr (car '`α`))))`	
`(cddddr '`α`)`	$\Longrightarrow$	`(cdr (cdr (cdr (cdr '`α`))))`	

For example, `cadr` gets the second element of a list and `cddr` removes the first two elements of a list. Thus usefulness of these abbreviations probably explains why the names '`car`' and '`cdr`' have persisted.

Exercise 1.5.2.1 Show how the following LISP expressions are rewritten when they are evaluated by a LISP interpreter. E.g.:

`(car (cdr '(a b c d))` $\Longrightarrow$ `(car '(b c d))` $\Longrightarrow$ `'b`

(a) `(car '(1 2))`
(b) `(cdr '(1 2))`
(c) `(car '(1 . 2))`
(d) `(cdr '(1 . 2))`
(e) `(car (cons 2 '(b)))`
(f) `(cdr (cons 2 '(b)))`
(g) `(car (cons 2 'b))`
(h) `(cdr (cons 2 'b))`
(i) `(cdr (cdr ()))`
(j) `(car (car '((a b) c)))`
(k) `(car (cdr (car '((a b) c))))`
(l) `(cdr (cdr (car '((a b) c))))`
(m) `(cadadr '(x (a (b .  c)) ((e f)))))`
(n) `(cdadr '((x y z) (1 2 3)))`
(o) `(cons 'r (cons 2 (cons 'd 2)))`
(p) `(cons '* (cons 2 (cons 'z nil)))`
(q) `(cons (cons (cons 'r 2) 'd) 2)`
(r) `(cons '+ (cons (cons '* (cons 2 (cons 'x nil))) (cons 5 nil)))`

Exercise 1.5.2.2 Using the rewrite rule definitions of `my-nth`, `my-append`, and `my-reverse` in Figures 1.12, 1.13, and 1.14, show how the following LISP expressions are rewritten when they are evaluated by a LISP interpreter. E.g.:

```
(my-append '(a b) 'c) ==> (cons 'a (my-append '(b) 'c)
                      ==> (cons 'a (cons 'b (my-append '() 'c)))
                      ==> (cons 'a (cons 'b 'c))
                      ==> (cons 'a '(b . c))
                      ==> '(a b . c)
```

(a) `(my-nth 0 '(1 2 3))`
(b) `(my-nth 2 '(1 2 3))`
(c) `(my-nth 2 '(1))`
(d) `(my-append '(b) '(d))`
(e) `(my-append '(b) 'd)`
(f) `(my-reverse '(b) '(d))`
(g) `(my-reverse '(b c))`

1.5.3 Data Predicates

COMMONLISP defines various predicate functions that test data. We define some of these here by rewrite rules:

`(consp '`α`)`	$\Longrightarrow$	`nil`	
`(consp '(`β` . `γ`))`	$\Longrightarrow$	`t`	
`(eql '`α_1` '`α_2`)`	$\Longrightarrow$	`t`	if α_1 and α_2 print the same
	$\Longrightarrow$	`nil`	otherwise
`(eql '(`β` . `γ`) '`α`)`	$\Longrightarrow$	`nil`	
`(eql '`α` '(`β` . `γ`))`	$\Longrightarrow$	`nil`	
`(eql '(`β_1` . `γ_1`) '(`β_2` . `γ_2`))`	$\Longrightarrow$	undefined for now	
`(symbolp '`α`)`	$\Longrightarrow$	`t`	if $\alpha \in$ symbol
`(symbolp '`β`)`	$\Longrightarrow$	`nil`	if $\beta \notin$ symbol
`(numberp '`α`)`	$\Longrightarrow$	`t`	if $\alpha \in$ number
`(numberp '`β`)`	$\Longrightarrow$	`nil`	if $\beta \notin$ number

where $\alpha, \alpha_1, \alpha_2 \in$ atom, $\quad \beta, \beta_1, \beta_2, \gamma, \gamma_1, \gamma_2 \in$ s-expression

Figure 1.15 gives examples inputting these functions to a LISP interpreter. This figure also gives examples of the following functions which may be defined in terms of the above functions.

```
(consp 'a)
--->
NIL

(consp '(1 . 2))
--->
T

(eql 'a 'a)
--->
T

(eql 'a 'b)
--->
NIL

(eql 5 5)
--->
T

(eql 5 6)
--->
NIL

(eql 5 5.0)
--->
NIL

(eql 'a '(a . b))
--->
NIL

(eql '(a . b) 'a)
--->
NIL
```

```
(symbolp 'a)
--->
T

(symbolp '(a b))
--->
NIL

(symbolp '())
--->
T

(symbolp 5)
--->
NIL

(numberp '5a)
--->
NIL

(numberp '5)
--->
T

(atom 'a)
--->
T

(atom '(1 . 2))
--->
NIL

(atom nil)
--->
T
```

```
(atom '())
--->
T

(null nil)
--->
T

(null '())
--->
T

(null '(a . b))
--->
NIL

(null 'a)
--->
NIL

(listp '())
--->
T

(listp nil)
--->
T

(listp '(a . b))
--->
T

(listp 'a)
--->
NIL
```

Figure 1.15: List Data Predicate Examples

```
;; (my-member 'x '()) ===> nil
;; (my-member 'x  '(y . z))
;;       ===> '(y . z)               if (eql 'x 'y)
;;       ===> (my-member 'x 'z)      otherwise
;;
;; where x, y, z are s-expressions
;;
(defun my-member (x w)
  (cond
   ((null w) nil)
   ((eql x (car w)) w)
   (t (my-member x (cdr w)))))
--->
MY-MEMBER

(my-member 'a '(a b c))
--->
(A B C)

(my-member 'b '(a b c))
--->
(B C)

(my-member 'c '(a b c))
--->
(C)

(my-member 'x '(a b c))
--->
NIL
```

Figure 1.16: The `Member` Function

```
;; (my-equal 'u 'v) ===> (eql 'u 'v)  if either u or v
;;                                      is an atom
;; (my-equal '(w . x) '(y . z)) ===>
;;      (and (equal 'w 'y) (equal 'x 'z))    otherwise
;;
;; where u, v, w, x, y, z are s-expressions
;;
(defun my-equal (u v)
  (cond
   ((or (atom u) (atom v))
    (eql u v))
   (t
    (and (my-equal (car u) (car v))
         (my-equal (cdr u) (cdr v))))))
--->
MY-EQUAL

(my-equal 'a 'a)
--->
T

(my-equal '(a b) '(a b))
--->
T

(my-equal '(a (b c) 5) '(a (b c) 5))
--->
T

(my-equal '(a (b c) 5) '(a (b x) 5))
--->
NIL
```

Figure 1.17: The `Equal` Function

(null 'β) $\Longrightarrow$ (eql 'β nil)
(atom 'β) $\Longrightarrow$ (not (consp 'β))
(listp 'β) $\Longrightarrow$ (or (null 'β) (consp 'β))

where $\beta \in$ s-expression

Here two functions that perform boolean logic, '`not`' and '`or`' are used. These have the obvious definition, in so far as their use here is concerned.[21].

Examples of higher level functions that use these predicates are given in Figures 1.16, and 1.17. These examples show how the COMMONLISP `member`, and `equal` functions might be coded. In order to avoid conflict with the actual COMMONLISP functions, the functions in the examples are given the names `my-member`, and `my-equal`.

The `my-equal` function makes use of the COMMONLISP '`and`' function that performs boolean logic. This has the obvious definition, in so far as its use here is concerned.[22]

Exercise 1.5.3.1 Give rewrite rule definitions (<u>not</u> `defun`'s) for the following functions, using the basic list manipulation and data predicate functions we have described in this chapter:

(a) A function `count-conses` that returns the number of cons cells in an s-expression. E.g.:
```
(count-conses '((1 . 2) . (3 . (4 . nil)))) ===> 4
(count-conses '(1 (2 3) 4 5)) ===> 6
```

(b) A function `copy-types` that copies an s-expression replacing each atom other than `nil` with either `'symbol` or `'number` according to the type of the atom. E.g.:
```
(copy-types '(+ (x 8) (nil y (z)))) ===>
    '(symbol (symbol number)
             (nil symbol (symbol)))
```

[21] The '`or`' function is a LISP special function that evaluates its arguments from left to right until one returns true or all are evaluated. The first true (i.e., non-`nil`) argument value is the value of the '`or`' function call, or if no value is true, the function call returns `nil`

[22] The '`and`' function is a LISP special function that evaluates its arguments from left to right until one returns false (i.e., `nil`) or all are evaluated. If no value is false, the last value is the value of the '`and`' function call, or if a value is false, the function call returns `nil`

(c) A function `list-symbols` that produces a list of all the non-`nil` symbols in an s-expression. If symbols are duplicated in the s-expression, they are duplicated in the list of symbols. The order of the symbols in the list of symbols should be the same as the order of the symbols in the usual printed representation of the s-expression. E.g.:
```
(list-symbols '(1 x (2 y (x)) z)) ===> '(x y x z)
```

(d) A function `inside` that returns true if and only if a given atom is somewhere inside an s-expression. `nil` is treated like any other atom, and counts as inside the s-expression if it appears as any `car` or `cdr`. E.g.:
```
(inside 'x '(+ (* 8 x) y)) ===> true
(inside 'z '(+ (* 8 x) y)) ===> false
```

(e) A function `bubble` that takes a number and a sorted list of numbers and returns the sorted list with the number included. (This function is part of a bubble sort.) E.g.:
```
(bubble 5 '(1 3 8 9)) ===> '(1 3 5 8 9)
(bubble -2 '(1 3 8 9)) ===> '(-2 1 3 8 9)
(bubble 10 '(1 3 8 9)) ===> '(1 3 8 9 10)
(bubble 5 '()) ===> '(5)
```

(f) A function `bubble-sort` that takes a list of numbers and sorts it. (You are <u>required</u> to use the '`bubble`' function above.) E.g.:
```
(bubble-sort '(5 1 8 9 3)) ===> '(1 3 5 8 9)
(bubble-sort '(3 9 -2 1 8)) ===> '(-2 1 3 8 9)
(bubble-sort '()) ===> '()
```

(g) A function `remove-dups` that takes a list of atoms and removes duplicate atoms. The order of the list is preserved and all but the last copy of each atom is removed. E.g.:
```
(remove-dups '(a 8 b a 3 8)) ===> '(b a 3 8)
(remove-dups '(a b c 8 8 c b)) ===> '(a 8 c b)
(remove-dups '()) ===> '()
```

(h) A function `my-replace` that replaces every occurrence of a symbol in a first s-expression by a given second s-expression. E.g.:

```
(my-replace 'x '(3 x z) 'y) ===> '(3 y z)
(my-replace 'x 'x 'y) ===> 'y
(my-replace 'x 'z 'y) ===> 'z
(my-replace 'x '(3 (* x y) z) '(+ 3 4)) ===>
          '(3 (* (+ 3 4) y) z)
(my-replace 'x 'x '(+ 3 4)) ===> '(+ 3 4)
(my-replace 'x '9 '(+ 3 4)) ===> '9
(my-replace 'x '(x . y) '9) ===> '(9 . y)
```

(i) A function `match` that takes two s-expressions and returns true if they 'match'. Two atoms match if they are `eql` or one is the symbol '?'. Two cons-cells match if their car's and cdr's match respectively. A symbol and a cons-cell match only if the symbol is '?'. Nothing else matches. E.g.:

```
(match '(x y) '(x y)) ===> t
(match '(x z) '(x y)) ===> nil
(match '(x ?) '(x y)) ===> t
(match '(x 6) '(x ?))  ===> t
(match '(x ? ?) '(x y)) ===> nil
(match '(9 ? . ?) '(9 x y z)) ===> t
```

(j) A function `my-union` that takes two lists of atoms and returns the second list with all the elements from the first list that are not in the second list prepended. The order of elements is the lists is preserved. E.g.:

```
(my-union '(1 2 3) '(2 3 4)) ===> '(1 2 3 4)
(my-union '(1 2) '(3 4)) ===> '(1 2 3 4)
(my-union '(x y) '(x y)) ===> '(x y)
(my-union '(x y z w) '(y x)) ===> '(z w y x)
```

(k) A function `my-intersection` that takes two lists of atoms and returns the order-preserving sublist of the first list that contains all elements of the first list that are also in the second list. E.g.:

```
(my-intersection '(1 2 3) '(2 3 4)) ===> '(2 3)
(my-intersection '(1 2) '(3 4)) ===> '()
(my-intersection '(x y) '(y x)) ===> '(x y)
(my-intersection '(x y z w) '(y x)) ===> '(x y)
```

(l) A function `find-sublists` that takes a list of atoms and returns a list of all the order-preserving sublists of the input list. E.g.:
`(find-sublists '(1 2)) ===> '((1 2) (1) (2) ())`
`(find-sublists '()) ===> '(())`

Exercise 1.5.3.2 Give the function definitions (i.e., `defun`'s) for the functions of the last exercise, using the basic list manipulation and data predicate functions we have described in this chapter:

(a)	`count-conses`	(g)	`remove-dups`
(b)	`copy-types`	(h)	`my-replace`
(c)	`list-symbols`	(i)	`match`
(d)	`inside`	(j)	`my-union`
(e)	`bubble`	(k)	`my-intersection`
(f)	`bubble-sort`	(l)	`find-sublists`

Exercise 1.5.3.3 Using definitions from either of the last two exercises, draw the call tree for the execution of the following. Include calls to `cons`, but not to any other function defined by COMMONLISP. Label nodes with the called function name and the arguments but <u>not</u> with the returned value.

```
(a1) (count-conses '((1 . 2) . (3 . (4 . nil))))
(a2) (count-conses '(1 (2 3) 4 5))
(b1) (copy-types '(+ (x 8) (nil y (z))))
(c1) (list-symbols '(1 x (2 y (x)) z))
(d1) (inside 'x '(+ (* 8 x) y))
(d2) (inside 'z '(+ (* 8 x) y))
(e1) (bubble 5 '(1 3 8 9))
(e2) (bubble 5 '())
(f1) (bubble-sort '(5 1 8 9 3))
(f2) (bubble-sort '())
(g1) (remove-dups '(a b c 8 8 c b))
(g2) (remove-dups '())
(h1) (my-replace 'x '(3 x z) 'y)
(h2) (my-replace 'x '(3 (* x y) z) '(+ 3 4))
(i1) (match '(x y) '(x y))
(i2) (match '(x z) '(x y))
(i3) (match '(x (? . z)) '(x (y . ?)))
(i4) (match '(x (w . z)) '(x (y . ?)))
```

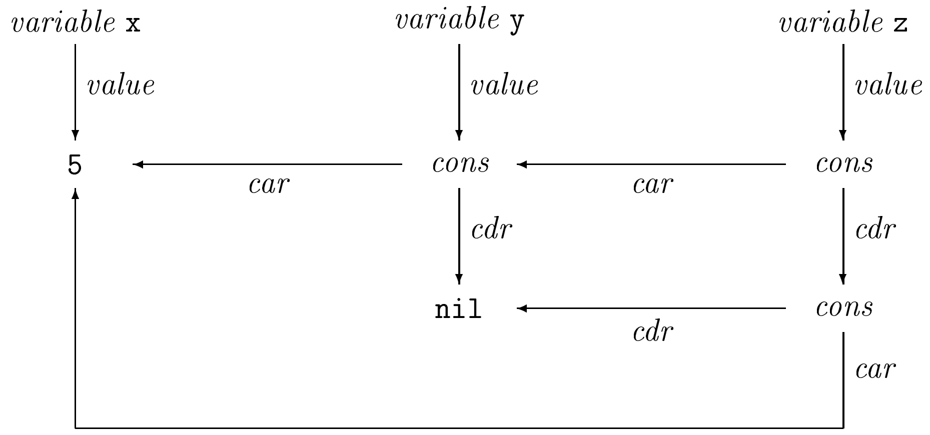


Figure 1.18: Example LISP Data Structure

```
(j1)  (my-union '(1 2 3) '(2 3 4))
(j2)  (my-union '(1 2) '(3 4))
(k1)  (my-intersection '(1 2 3) '(2 3 4))
(k2)  (my-intersection '(1 2) '(3 4))
(l1)  (find-sublists '(1 2))
(l2)  (find-sublists '())
```

1.5.4 The LISP Memory Model

LISP data can be represented by a labeled graph which is an abstract representation of the way the data is actually stored in computer memory. In this graph each symbol, number, cons-cell, and variable is a node; and each LISP value is an arrow pointing at one of these nodes. An example of such a graph is given in Figure 1.18.

We will refer to such a graph as a '**LISP memory graph**'. In this graph each atom is represented by a node labeled by the value of the atom. These atom nodes are '**leaf**' nodes, meaning that these nodes are <u>not</u> the origins of any arrows. Each cons-cell is represented by a node labeled 'cons' from which two arrows labeled '`car`' and '`cdr`' originate. Each variable is represented by a node labeled with 'variable' and the name of the variable, from which one arrow labeled 'value' originates. Any LISP value is represented by an arrow pointing at an atom or cons-cell.

Furthermore, each atom has only one unique node in the entire graph with its label, so all values that are the same atom point to the same node in the graph. Similarly no two variable nodes can have the same label. Cons-cells, on the other hand, can have many duplicates in the same graph.

So far the only way that we have mentioned to introduce a LISP variable is by giving a name for an argument in a function definition. Another way is the LISP '`let*`' expression. The following `let*` expression produced the memory graph in Figure 1.18:

```
(let* (( x 5 )
       ( y (cons 5 nil) )
       ( z (cons y (cons x nil)) ))
   . . . )
```

This first creates a new variable `x` and assigns it the value `5`. In graphical terms, this means that the nodes labeled 'variable `x`' and '`5`' are created, and an arrow labeled 'value' is created from the 'variable `x`' node to the '`5`' node.

Actually, if a node labeled '`5`' already exists in the memory graph, it will be used instead of creating a new node labeled '`5`'. The node of an atom is created when the atom first needs to be represented in the memory graph.

If a node labeled 'variable `x`' already exists, this old 'variable `x`' node is temporarily erased, along with the arrow pointing from it, but both are remembered. When the `let*` statement finishes evaluating, then all the variable nodes it introduced are erased and discarded, and any remembered variable nodes they replaced are recreated.

After creating the 'variable `x`' node and its value arrow, the `let*` statement above evaluates the expression '`(cons 5 nil)`' which will become the value of the new `y` variable. The execution of the `cons` function creates a new 'cons' node with a `car` arrow pointing at the '`5`' node and a `cdr` arrow pointing at the '`nil`' node. The '`nil`' node, like all atom nodes, is created the first time it is needed. Then the `let*` expression creates a 'variable `y`' node and a 'value' arrow from it to the new 'cons' node.

Lastly the `let*` statement evaluates `(cons y (cons x nil))` which creates two new 'cons' nodes, and then the `let*` creates a 'variable `z`' node and a 'value' arrow from it to the last 'cons' node created. Evaluation of the expression `(cons x nil)` produces a new cons node whose `car` arrow points at the same node the `x` variable points at, and whose `cdr` points at the `nil` node. The second call to `cons` produces a new cons

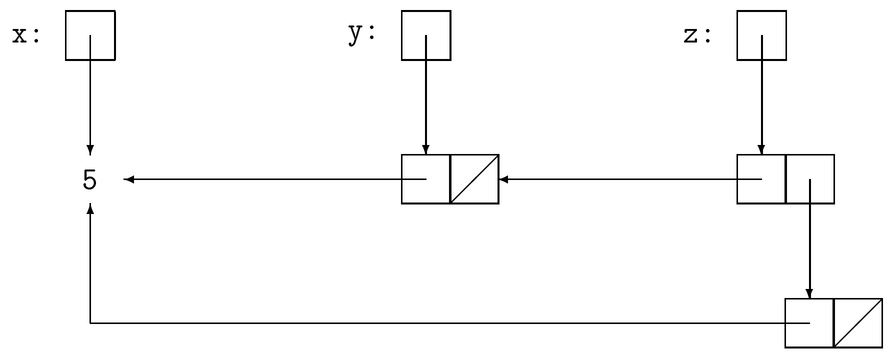


Figure 1.19: Box Notation Example

node that is pointed at by the `z` variable and that has a `car` arrow pointing at the same node the `y` variable points at and a `cdr` arrow pointing at the cons node created by `(cons x nil)`.

The values of variables, function arguments, and function values are all arrows pointing at atom or 'cons' nodes in the LISP memory graph. The `car` and `cdr` of a cons cell are also origins of such arrows. We say that a '**LISP value**' is represented by an arrow pointing at an atom or 'cons' node, and that LISP values can be stored in variables, passed as function arguments or function values, and stored in the `car` or `cdr` of a cons cell.

What we have described is an abstract but precise representation of how the data memory of the LISP interpreter works. This description is called the '**LISP memory model**': it is a model of how the memory operated on by LISP programs works. All programming languages have a memory model, and it is impossible to understand the programming language properly without understanding the memory model precisely.

Because LISP memory graphs are so useful, a shorthand notation has been developed for them. The memory graph in Figure 1.18 is abbreviated to make the graph in Figure 1.19. All arrow labels are omitted. Cons nodes are drawn as a pair of square boxes with the arrow from the left box being the `car` arrow, and the arrow from the right box being the `cdr` arrow. Variables are just square boxes labeled by the variable name. Because pointers to `nil` are so common and tend to clutter the drawing, they are abbreviated by drawing a diagonal line through the box from which they originate, as in ⧄. Unsurprisingly graphs drawn with this abbreviated notation are said to be drawn in **box notation**.

In LISP there is a correspondence between certain labeled graphs and s-expressions. When describing an algorithm, it is often more convenient to describe the behavior of the algorithm in terms of s-expressions than to describe its behavior in terms of labeled graphs, simply because its easier to write s-expressions than it is to draw labeled graphs. In particular, rewrite rules involving s-expressions are commonly used to describe algorithms that operate on labeled graphs.

The general case of the `let*` expression is:

$$
\begin{array}{l}
\texttt{(let* ((}v_1\ e_1\texttt{)} \\
\qquad\qquad\texttt{(}v_2\ e_2\texttt{)} \\
\qquad\qquad\ldots\texttt{)} \\
\quad e_0\texttt{)}
\end{array}
$$

This first evaluates expression e_1 to get a value, then creates variable v_1 and assigns it the value, then evaluates e_2 to get a value, then creates variable v_2 and assigns it the value, etc. Lastly the expression e_0 is evaluated to get a final value, all the variables that were created are destroyed, and the final value is returned as the value of the `let*` expression.

There is a variant of the `let*` expression, namely the `let` expression, that evaluates all the expressions e_1, e_2, ...before creating any variables. It then creates all the variables at once, assigning each the corresponding value. The `let` expression is not as useful as `let*` because the expressions that produce the variable values cannot reference previous variables in the same `let` expression.

The LISP memory model can be used to explain the `eql` function precisely. This function takes two LISP values and returns true if and only if they both point to the same node in the LISP memory graph.

Thus

```
(let* ((x 'a)
       (y 'a))
  (eql x y))
--->
T
```

```
(let* ((x (cons 'a nil))
       (y (cons 'a nil)))
  (eql x y))
--->
NIL
```

because there can be only one node labeled 'A' in any LISP memory graph, but two different calls to the `cons` function will produce two different cons cell nodes in the memory graph.

Similarly inside the context of the memory graph of Figure 1.19, the following hold:

```
(eql x 5) ---> t
(eql x (car y)) ---> t
(eql x (cdr y)) ---> nil
(eql x (car z)) ---> nil
(eql x (cdr z)) ---> nil
(eql x (car (car z))) ---> t
(eql x (car (cdr z))) ---> t
(eql y (car z)) ---> t
(eql y (car (car z))) ---> nil
(eql y (car (cdr z))) ---> nil
```

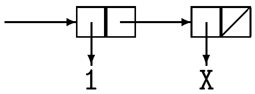
LISP memory graphs are special cases of '**object graphs**'. The data memory of many programs can be thought of as consisting of **objects** that are nodes in a graph. Each object has an **address** and a set of labeled **components**. Given the address of an object and the label of one of its components, it is possible to get the value of that component of the addressed object. It may also be possible to store new values in such a component.

Typically an object is a contiguous block of memory bytes with the object address being the address of the first byte. But this need not be the case.

Objects are represented as nodes in the object graph, which might be depicted as is the example in Figure 1.33. A component that stores addresses is represented by an arrow from the object it is part of to the object it addresses. The component label is used as the label of the arrow. Various labels may be put on the object node, including the names and values of the non-address components.

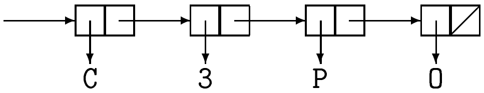
A variety of terminology is in use. Object components may be called **attributes**, **data members**, or **properties**. Addresses are often called **pointers**.

Exercise 1.5.4.1 For the following lisp memory model graphs, write the s-expression that represents the graph. For example:

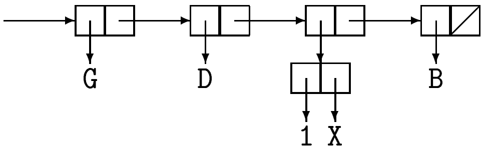


is represented by '`(1 x)`' (which is a simplification of '`(1 . (x . nil))`').

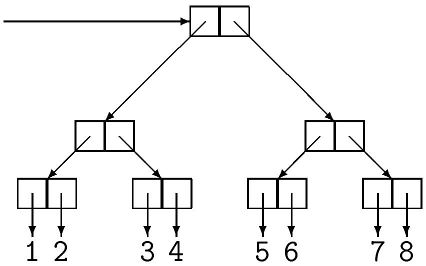
(a)



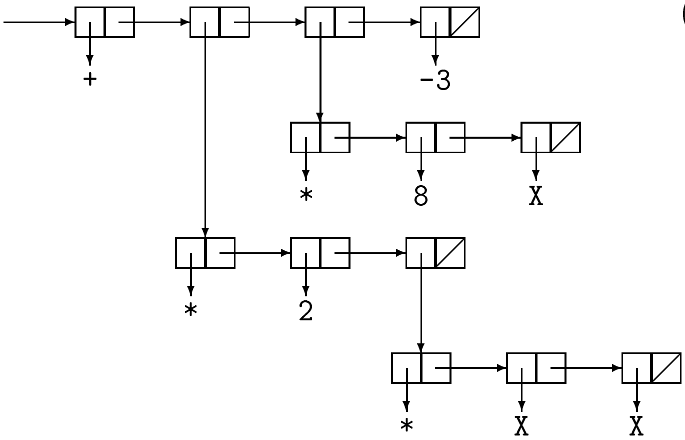
(b)



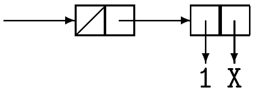
(c)



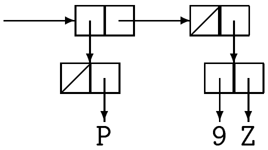
(d)



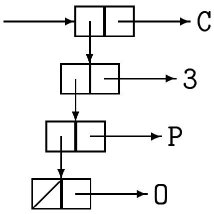
(e)



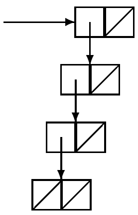
(f)



(g)



(h)



Exercise 1.5.4.2 Draw the lisp memory model graphs represented by the following LISP s-expressions (see previous exercise for examples):

(a)	`(+ x y x)`	(f)	`(+ (* x y) 4)`
(b)	`'x`	(g)	`((a . b) . (c . b))`
(c)	`(((x)))`	(h)	`(nil nil .   nil)`
(d)	`((a . b) c)`	(i)	`''x`
(e)	`(sqrt 3.5)`	(j)	`(member 'x '(1 2 1))`

1.6 Procedure Abstraction

In this section we show how LISP can treat functions as if they were data. Functions are sometimes called procedures, and a procedure that has no name and is treated like data is called a '**procedure abstraction**'.

1.6.1 Lambda, Funcall, and #'

A function can be represented as data by notation such as:

```
#'(lambda (n) (* n n))
```

which defines a function that behaves like the `square` function defined in Figure 1.3. Such a function has no name, and has historically been called a 'procedure abstraction'. We will more often call it a '**closure**', for reasons that will become apparent in Section 1.6.3.

In fact, what:

```
(defun square (n) (* n n))
```

does is to compute the closure above and associate it with the symbol `square` in a way that permits `square` to be used as the function name of the closure.

Since a closure is a function, it can be called with arguments. In LISP this is done by the `funcall` function, as in:

`(funcall #'(lambda (n) (* n n)) 5)` $\Longrightarrow$ `(* 5 5)` $\Longrightarrow$ `25`

Since `defun` associates a closure with a name, there should be some way of retrieving the closure given the name. This can be done by `#'`, used as in '`#'square`'. Thus:

$$\texttt{(funcall \#'square 5)} \Longrightarrow \texttt{(* 5 5)} \Longrightarrow \texttt{25}$$

provided `square` was defined by the above `defun`.

`#'` is like `'` except that it implies the quoted s-expression should be a closure, and on input `#'` is equivalent to a call to a special function named '`function`' instead of to a special function named '`quote`'. That is:

$$\texttt{'}\beta \equiv \texttt{(quote } \beta\texttt{)}$$
$$\texttt{\#'}\beta \equiv \texttt{(function } \beta\texttt{)}$$

where $\beta \in$ s-expression

If `#'` quotes a list that begins with the symbol `lambda`, the list is interpreted as a closure definition. This list (e.g., '`(lambda (n) (* n n))`') consists of the symbol '`lambda`', followed by a list of argument names (e.g., '`(n)`'), followed by the expression to be evaluated to produce the result of the closure when it is used as a function (e.g., '`(* n n)`'). This last expression is called the '**body**' of the closure. Thus:

closure-expression	::=	`#'(lambda` argument-list body`)`
argument-list	::=	`(` argument-name* `)`
argument-name	::=	symbol
body	::=	s-expression

If `#'` quotes a symbol, the effect is to retrieve the closure whose function name is the symbol. It is an error if there is no such closure.

Figure 1.20 gives examples involving `defun`, `funcall`, `#'`, and `function`. Evaluation of a **closure expression** returns a closure. Exactly what a closure is depends upon the particular LISP interpreter being used. For the LISP interpreter in the figure, a closure is a new kind of LISP memory object, in addition to numbers, symbols, and cons cells. This new kind of object prints as '`#<CLOSURE ...>`', and contains a number of parts that print, including the argument list and body, and in the case of closures defined by `defun`, the function name associated with the closure. No matter how a closure prints, its effect when passed to `funcall` is to assign values to the arguments in the argument list and then to evaluate the body and return the resulting value.

Figure 1.21 shows how the COMMONLISP `mapcar` function might be coded. The `mapcar` function takes a function and a list as arguments, and returns the list with the function applied to each element of the list. Thus:

(mapcar #'oddp '(1 2 3 4 5)) $\Longrightarrow$ '(t nil t nil t)

In the rest of this book we will use the word '**function**' to refer to either a function name or a closure.

Exercise 1.6.1.1 Give rewrite rule definitions (<u>not</u> `defun`'s) for the following functions, using the basic list manipulation and data predicate functions we have described in this chapter:

(a) A function `tested-sublist` that returns the order-preserving sublist of a list (the first argument) consisting of all elements of that argument that satisfy a predicate function (the second argument). E.g.:

```
(tested-sublist '(1 2 3 4 5) #'oddp) ===> '(1 3 5)
(tested-sublist '(1 x 3 y 5) #'symbolp) ===> '(x y)
(tested-sublist '(1 5 9 3 7) #'(lambda (n) (> n 4)))
        ===> '(5 9 7)
```

(b) A function `look-into` that returns true if and only if some non-nil atom inside an s-expression (the first argument) satisfies a predicate (the second argument). E.g.:

```
(look-into '(1 8 (2 4 (x)) 9) #'symbolp) ===> t
(look-into '(1 8 (2 4 (7)) 9) #'symbolp) ===> nil
(look-into '(5 8 (2 4 (3)) 9) #'(lambda (n) (< n 3)))
        ===> t
(look-into '(5 8 (7 4 (3)) 9) #'(lambda (n) (< n 3)))
        ===> nil
```

(c) A function `poke` that returns true if and only if two consecutive elements of a list (the first argument) satisfy a two argument predicate (the second argument). The function is undefined if the list has less than two elements. E.g.:

```
(poke '(8 2 3) #'>) ===> t
(poke '(2 2 3) #'>) ===> nil
(poke '(8 2 3 4 4 8 3 1) #'=) ===> t
(poke '(8 2 3 4 5 8 3 1) #'=) ===> t
```

```
(defun square (n) (* n n))
--->
SQUARE

#'square
--->
#<CLOSURE SQUARE (N) (DECLARE (SYSTEM::IN-DEFUN SQUARE))
  (BLOCK SQUARE (* N N))>

#'(lambda (n) (* n n))
--->
#<CLOSURE :LAMBDA (N) (* N N)>

(function (lambda (n) (* n n)))
--->
#<CLOSURE :LAMBDA (N) (* N N)>

(funcall #'(lambda (n) (* n n)) 3)
--->
9

(funcall #'square 3)
--->
9

(funcall (function square) 3)
--->
9
```

Figure 1.20: `Defun`, `Funcall`, and `#'`

```
;; (my-mapcar #'f '()) ===> '()
;; (my-mapcar #'f '(x . y)) ===>
;;      (cons (funcall #'f 'x) (my-mapcar  #'f 'y))
;;
;; where f, x, y are s-expressions
;;
(defun my-mapcar (f w)
  (cond
   ((null w) nil)
   (t (cons (funcall f (car w))
            (my-mapcar f (cdr w))))))
--->
MY-MAPCAR

(my-mapcar #'square '())
--->
NIL

(my-mapcar #'square '(5))
--->
(25)

(my-mapcar #'square '(5 3))
--->
(25 9)

(my-mapcar #'oddp '(1 2 3 4 5))
--->
(T NIL T NIL T)
```

Figure 1.21: `Mapcar`

(d) A function `fold` that folds up a list of values (the first argument) into a single value by applying a two argument function (the second argument) as an operator to successive elements of the list. Informally:
`(fold '(1 2 3 4) #'f) ===> (f (f (f 1 2) 3 ) 4)`
The `fold` function is undefined if the list has less than two elements. Some example uses are:
```
(fold '(1 2 3) #'+) ===> (+ (+ 1 2) 3) ===> 6
(fold '(1 2 3 4) #'*) ===> (* (* (* 1 2) 3) 4) ===> 24
(fold '(1 2 3 4) #'cons) ===> '(((1 . 2) . 3) . 4)
```

Exercise 1.6.1.2 Give the function definitions (i.e., `defun`'s) for the functions of the last exercise, using the basic list manipulation and data predicate functions we have described in this chapter:

(a) `tested-sublist` (c) `poke`
(b) `look-into` (d) `fold`

Exercise 1.6.1.3 Using definitions from either of the last two exercises, draw the call tree for the execution of the following. Include calls to `cons` and `funcall` but not to any other function defined by COMMONLISP. Label nodes with called function name and arguments but <u>not</u> with the returned value.

(a) `(tested-sublist '(1 2 3 4) #'oddp)`
(b) `(look-into '((3 . 4) . 4) #'oddp)`
(c) `(look-into '((8 . 4) . 6) #'oddp)`
(d) `(poke '(2 8 8 6) #'=)`
(e) `(poke '(2 8 4 6) #'=)`
(f) `(fold '(2 8 4 6) #'+)`

1.6.2 Examples Involving Summing

Figure 1.22 gives code for the `sum` function that sums a consecutive series of numbers, and code for the `sum-square` function that sums the squares of a consecutive series of numbers. Thus:

$$\begin{array}{lcl} \texttt{(sum } m\ n\texttt{)} & \Longrightarrow & m + (m+1) + (m+2) + \ldots + (n-1) + n \\ \texttt{(sum-square } m\ n\texttt{)} & \Longrightarrow & m^2 + (m+1)^2 + (m+2)^2 + \ldots + (n-1)^2 + n^2 \end{array}$$

where m, $n \in$ number

```
;; (sum m n) ===> 0                           if m > n
;;           ===> (+ m (sum (+ m 1) n))   otherwise
;;
;; where m, n are numbers
;;
(defun sum (m n)
  (cond
   ((> m n) 0)
   (t (+ m (sum (+ m 1) n)))))
--->
SUM

(sum 1 3)
--->
6

(sum 2 5)
--->
14

;; (sum-square m n)
;;     ===> 0                           if m > n
;;     ===> (+ (square m)
;;              (sum-square (+ m 1) n))   otherwise
;;
;; where m, n are numbers
;;
(defun sum-square (m n)
  (cond
   ((> m n) 0)
   (t (+ (square m) (sum-square (+ m 1) n)))))
--->
SUM-SQUARE

(sum-square 1 3)
--->
14

(sum-square 2 5)
--->
54
```

Figure 1.22: Sum and Sum-Square

```
;; (sum-term #'f m n s)
;;    ===> 0                                if m > n
;;    ===> (+ (funcall #'f m)
;;            (sum-term #'f (+ m s) n s))   otherwise
;;
;; where f is an s-expression,  m, n, s are numbers
;;
(defun sum-term (f m n s)
  (cond
   ((> m n) 0)
   (t (+ (funcall f m) (sum-term f (+ m s) n s)))))
--->
SUM-TERM


(sum 1 3)
--->
6

(sum-term #'(lambda (n) n) 1 3 1)
--->
6

(sum-square 2 5)
--->
54

(sum-term #'square 2 5 1)
--->
54

; Need to compile for subsequent examples:

(compile 'sum-term)
--->
SUM-TERM
```

Figure 1.23: `Sum-Term`

```
(defun integral (f x1 x2 dx)
  (sum-term
   #'(lambda (x) (* dx (funcall f (+ x (* 0.5 dx)))))
   x1 x2 dx))
--->
INTEGRAL

;; Area under straight line y = x from x = 0.0 to
;; x = 4.0:
;;
(integral #'(lambda (x) x) 0.0 3.999 0.01)
--->
7.999999999999961

(integral #'(lambda (x) x) 0.0 3.9999 0.001)
--->
7.999999999999506

;; Area under quarter circle of unit radius from
;; x = 0.0 to x = 1.0:
;;
(integral
  #'(lambda (x) (sqrt (- 1.0 (* x x))))
  0.0 0.999 0.01)
--->
0.7854842144750017

(integral
  #'(lambda (x) (sqrt (- 1.0 (* x x))))
  0.0 0.9999 0.001)
--->
0.7854008862282249

(/ pi 4)
--->
0.785398163397448309 63L0
```

Figure 1.24: `Integral`

```
(defun sum-pi (m)
  (* 8 (sum-term
        #'(lambda (n) (/ 1.0 (* n (+ n 2))))
        1 (* 4 m) 4)))
--->
SUM-PI

(sum-pi 10)
--->
3.0916238066678385

(sum-pi 100)
--->
3.1365926848388166

(sum-pi 1000)
--->
3.141092653621043

pi
--->
3.141592653589793238 5L0
```

Figure 1.25: Computing π by Series

Here the right sides of the rewrite rules assume for simplicity that $n - m$ is an integer, although the functions defined in Figure 1.22 are more general.

These two functions can be replaced by the single `sum-term` function that applies some other function f to each number in a consecutive series of numbers, and sums the results. Thus:

(sum-term #'f m n s)
$$\Longrightarrow \quad f(m) + f(m+s) + f(m+2s) + \ldots + f(n-s) + f(n)$$
where m, n, $s \in$ number; $f \in$ function-name

where again we oversimplify and assume that $n - m$ is an integral multiple of s. Thus:

(sum m n)	$\Longrightarrow$	(sum-term #'(lambda (x) x) m n 1)
(sum-square m n)	$\Longrightarrow$	(sum-term #'square m n 1)

where m, $n \in$ number

An implementation and example uses of `sum-term` is given in Figure 1.23.

At the end of this figure the `sum-term` function is '**compiled**' by calling the '`compile`' function with the name of the function to be compiled as an argument. The closure associated with the name `sum-term` is changed from being an interpreted closure to being a compiled closure. After this is done, `sum-term` will run much faster, because compiled code runs much faster than interpreted code.

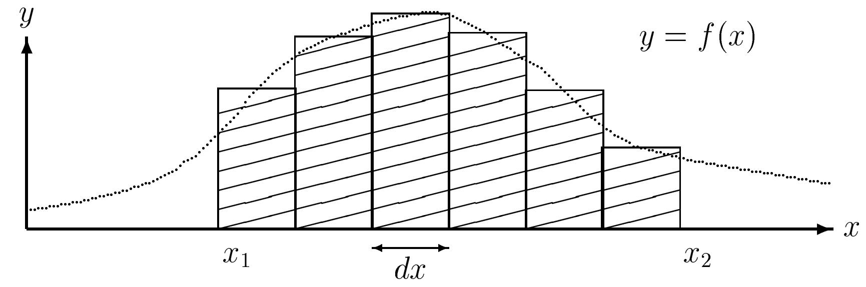
One interesting use of `sum-term` is to compute integrals. Here we define an integral to be:

the area under the curve '$y = f(x)$' from x_1 to x_2

or in mathematical symbols

$$\int_{x_1}^{x_2} f(x)dx$$

A pictorial depiction of this integral suggesting how to compute it is:



The integral is computed by dividing the line from x_1 to x_2 into intervals of width dx, and for each interval $[x, x+dx]$, drawing the rectangle whose base is the interval and whose height is $f(x+\frac{dx}{2})$. The integral (i.e. the area) is then approximately the sum of the areas of all these rectangles. The smaller we choose dx to be, the more accurate the approximation will be.

We can render this into LISP by defining a function `integral` that computes approximate integrals using the rewrite rule:

(integral #'f x_1 x_2 dx) $\Longrightarrow$
    (sum-term #'(lambda (x) (* dx (funcall #'f (+ x (* 0.5 dx)))))
        x_1 x_2 dx)
where $f \in$ s-expression (denoting a closure), x_1, x_2, $dx \in$ numbers

The `#'(lambda (x) ...)` closure in this rule computes the area of the rectangle with base $[x, x+dx]$ and height $f(x+\frac{dx}{2})$.

Figure 1.24 gives an implementation and example uses of `integral`. The first example computes the area under a straight line, and for this case the approximation is exact regardless of the value of dx (modulo the limited precision of floating point numbers). The second example computes the area in 1/4'th of the unit circle. The expected result is $\pi/4$, and the result is accurate to 0.000003 with $dx = 0.001$. In all cases one must be careful to set x_2 to be just <u>before</u> the right end of the base of the last rectangle, so one does not get an extra rectangle, as this would cause a much larger error than one would otherwise have.

Another interesting use of `sum-term` is to compute π by using the infinite series:

$$\frac{\pi}{8} = \frac{1}{1\cdot 3} + \frac{1}{5\cdot 7} + \frac{1}{9\cdot 11} + \frac{1}{13\cdot 15} + \frac{1}{17\cdot 19} + \cdots$$

A LISP function to compute an approximation to π by summing the first m terms of this series can be defined by the rewrite rule:

```
(sum-pi m) ⟹
      (* 8 (sum-term #'(lambda (n) (/ 1.0 (* n (+ n 2))))
                     1 (* 4 m) 4))
```
where $m \in$ integer, $m \geq 1$

Figure 1.25 gives an implementation and example uses of `sum-pi`. As can be seen, the error with $1,000$ terms is about 0.0005.

Exercise 1.6.2.1 Write s-expressions using `sum-term` that will compute the following:

(a) The sum of all cubes from 1 to an integer $\mathtt{n} \geq 1$.
(b) The squares of the odd integers in the sequence m, m+1, ..., n for integers $0 \leq \mathtt{m} \leq \mathtt{n}$.
(c) The sum of $(\frac{1}{2})^k$ for k from 1 to n, for some integer $\mathtt{n} \geq 1$.

1.6.3 Streams

A **stream** is a possibly infinite list. In this section we will implement a stream as a cons-cell whose car is the first element of the stream and whose cdr is a closure that takes no arguments and when called returns the rest of the stream.

Thus a stream is defined by the grammar:

empty-stream	::=	`nil`
non-empty-stream	::=	`(` first-element `.` rest-of-stream `)`
first-element	::=	s-expression
rest-of-stream	::=	closure

Operations on streams are define by:

```
(cons 'α 'φ)            ⟹  '(α . φ)
(stream-car '(α . φ))   ⟹  'α
(stream-car '())        ⟹  nil
(stream-cdr '(α . φ))   ⟹  (funcall 'φ)
(stream-cdr '())        ⟹  nil
```

where $\alpha \in$ s-expression, $\phi \in$ closure

```
;; (stream-car '()) ===> nil
;; (stream-car '(x . f)) ===> 'x
;;
;; where x, f are s-expressions
;;
(defun stream-car (w) (car w))
--->
STREAM-CAR

;; (stream-cdr '()) ===> nil
;; (stream-cdr '(x . f)) ===> (funcall 'f)
;;
;; where x, f are s-expressions
;;
(defun stream-cdr (w)
  (cond
   ((null w) nil)
   (t (funcall (cdr w)))))
--->
STREAM-CDR

(defun integers-from (n)
  (cons n #'(lambda () (integers-from (+ n 1)))))
--->
INTEGERS-FROM

(setf natural-numbers (integers-from 0))
--->
(0 . #<CLOSURE :LAMBDA NIL (INTEGERS-FROM (+ N 1))>)

(stream-cdr natural-numbers)
--->
(1 . #<CLOSURE :LAMBDA NIL (INTEGERS-FROM (+ N 1))>)

(stream-cdr (stream-cdr natural-numbers))
--->
(2 . #<CLOSURE :LAMBDA NIL (INTEGERS-FROM (+ N 1))>)

(stream-cdr (stream-cdr (stream-cdr natural-numbers)))
--->
(3 . #<CLOSURE :LAMBDA NIL (INTEGERS-FROM (+ N 1))>)
```

Figure 1.26: Stream Examples

Figure 1.26 gives an implementation of `stream-car` and `stream-cdr` and an example involving the `integers-from` function. This last function returns the stream of consecutive integers starting from a given value n, and is defined by the rule:

(integers-from n) $\Longrightarrow$ '(n . ϕ)
    if #'(lambda () (integers-from (+ n 1))) $\Longrightarrow \phi$

where $n \in$ integer; $\phi \in$ closure

and the LISP code:

```
(defun integers-from (n)
  (cons n #'(lambda () (integers-from (+ n 1)))))
```

When the `integers-from` function computes the `#'(lambda ...)` closure, something very special is done with the variable `n`. Because `n` is used in the body `(integers-from (+ n 1))` of the closure, the variable `n` is attached to the closure, and thus is not destroyed, as it would normally be, when the `integers-from` function execution that computed the closure finishes.

Consider the specific example of a call to `integers-from` with argument `5`. During the function execution which evaluates this call, the variable `n` exists and has the value `5`. When the `#'(lambda ...)` closure is computed, the variable `n` with the value `5` is attached to the closure. When the `integers-from` function execution finishes, the variable `n` is not destroyed as it would normally be. It remains attached to the closure, but it is made invisible to everything but the closure. Much later when the closure is called, it uses the variable `n` attached to it, which has the value `5`. The evaluation of the body of the closure, `(integers-from (+ n 1))`, with this variable and its value, causes a call to `integers-from` with the argument `(+ 5 1)`$\Longrightarrow$`6`, which in turn returns the stream of integers beginning with `6`.

Thus a call to `integers-from` with argument `5` produces a cons-cell whose car is `5` and whose cdr is a closure that when called produces the result of a call to `integers-from` with argument `6`.

The reason a **closure** is called a 'closure' is because it captures variables from the environment in which the closure was created, so the closure does not need to use variables in the environment in which the closure is later called. Thus the closure is 'self-contained', or 'closed', and does not need variables from outside itself when it is called.

Exercise 1.6.3.1 If the code:

```
(setf x (let ((y nil))
              (cons #'(lambda () y)
                    #'(lambda (z) (setf y z)))))
```

is evaluated first, then what are the values of the following s-expressions if they are evaluated in the order given.[23]

(a) `(funcall (car x))`
(b) `(funcall (cdr x) 6)`
(c) `(funcall (car x))`
(d) `(funcall (cdr x) 7)`
(e) `(funcall (car x))`

1.7 Iteration and Macros

In programming the term '**iterate**' is used to mean 'to repeat a loop'. We prefer to use recursion rather than iteration in this book, but we will nevertheless introduce COMMONLISP iterative programming in this section. We will also introduce COMMONLISP macros and show how to use macros to define iterative programming constructs in terms of recursion.

1.7.1 Iteration

COMMONLISP provides for loops with the '`do`' expression which has the form:

$$\begin{array}{l}\texttt{(do ((}v_1\ i_1\ n_1\texttt{)}\\ \qquad\quad \texttt{(}v_2\ i_2\ n_2\texttt{)}\\ \qquad\quad \ldots\texttt{)}\\ \quad\ \texttt{(}e_t\ e_r\texttt{))}\end{array}$$

This executes the algorithm:

1. Evaluate all the expressions i_1, i_2, ... (the **initial values**).

[23] `(setf` v e`)` sets the variable v to the value of the expression e.

2. Create the variables v_1, v_2, $\ldots$ and assign them the values obtained from evaluating i_1, i_2, $\ldots$.

3. Evaluate the expression e_t (the **test**) and if false go to the next step to continue the loop. Otherwise, if true, evaluate e_r (the **result**) and save the value, destroy the variables v_1, v_2, $\ldots$, and terminate the loop returning the saved result value as the value of the '`do`' expression.

4. Evaluate all the expressions n_1, n_2, $\ldots$ (the **next values**).

5. Assign the variables v_1, v_2, $\ldots$ the values obtained from evaluating n_1, n_2, $\ldots$.

6. Go to step 3 to perform the next loop iteration.

The `sum` function in Section 1.6.2 above can be written using a `do` loop instead of recursion:

```
(do-sum m n) ⟹
          (do ((i m (+ i 1))
               (sum 0 (+ sum i)))
            ((> i n) sum))
```

where m, $n \in$ number

Here, in the first iteration of the loop, `i` is assigned the value m, and in each subsequent iteration `i` takes the values $m+1$, $m+2$, $\ldots$. The variable `sum` is assigned the value `0` for the first iteration, and in subsequent iterations `sum` takes the values $0+m$, $0+m+(m+1)$, $0+m+(m+1)+(m+2)$, $\ldots$. On the first iteration with `i` $> n$ the loop stops, and the value of `sum` is returned.

This code is presented to the LISP interpreter in Figure 1.27.

A `do` expression is precisely equivalent to a recursive function. Figure 1.27 gives the results of compiling the `do-sum do` expression into equivalent code that is based on a closure that calls itself. The variables of this closure are mostly just the variables of the `do` expression. The closure is initially called with the initial values of these variables as arguments. The closure evaluates the test expression to determine whether to terminate or continue the loop. To continue, the closure calls itself with the next values of the variables to perform another iteration. To terminate, the closure evaluates the result expression and returns its value.

```
;; Sum with DO:

(defun do-sum (m n)
  (do ((i m (+ i 1))
       (sum 0 (+ sum i)))
      ((> i n) sum)))
--->
DO-SUM

(do-sum 1 3)
--->
6

(sum 1 3)
--->
6

(do-sum 2 5)
--->
14

(sum 2 5)
--->
14
```

```
;; Sum with DO compiled into
;; a closure:

(defun compiled-do-sum (m n)

  ;; Hand compilation of:
  ;; (do ((i m (+ i 1))
  ;;      (sum 0 (+ sum i)))
  ;;     ((> i n) sum))
  (let ((loop-f
         #'(lambda (lf i sum)
             (cond
              ((> i n) sum)
              (t (funcall lf lf
                          (+ i 1)
                          (+ sum i)))))))
       (funcall loop-f loop-f m 0)))
--->
COMPILED-DO-SUM

(compiled-do-sum 1 3)
--->
6

(sum 1 3)
--->
6

(compiled-do-sum 2 5)
--->
14

(sum 2 5)
--->
14
```

Figure 1.27: Summing with Do

There is one technical problem: how does the closure name itself so it can call itself. In the compilation we give, this is solved by adding an extra first argument to the closure which in use is always equal to the closure itself.

Exercise 1.7.1.1 Give iterative, non-recursive function definitions (i.e., `defun`'s) for the following functions using COMMONLISP '`do`':

(a) The `power` function of Figure 1.8.
(b) The `my-nth` function of Figure 1.12.
(c) The `my-reverse` function of Figure 1.14.
(d) The `my-member` function of Figure 1.16.
(e) The Fibonacci function `f` of Exercise 1.4.7.1.

Exercise 1.7.1.2 For each of the below functions, explain why it is hard to give an iterative, non-recursive function definition (i.e., `defun`) using COMMONLISP '`do`'.

(a) The `my-append` function of Figure 1.13.
(b) The `my-equal` function of Figure 1.17.

Exercise 1.7.1.3 [Hard] For each of the below functions, suggest a way of writing an iterative, non-recursive function definition (i.e., `defun`) using COMMONLISP '`do`' in spite of the difficulties.

(a) The `my-append` function of Figure 1.13.
(b) The `my-equal` function of Figure 1.17.

1.7.2 The Backquote Notation and Macros

Suppose COMMONLISP had no '`do`' expression and we wanted to add one. How could we manage this?

To answer this question, we will investigate how to tell the COMMONLISP interpreter to implement the rewrite rule:

```
(my-do ((v1 i1 n1)
        (v2 i2 n2))
   (test result))
==>
(let ((loop-f
       #'(lambda (lf v1 v2)
           (cond (test result)
                 (t (funcall lf lf n1 n2))))))
     (funcall loop-f loop-f i1 i2)))
```

where v_1, $v_2 \in$ variable, i_1, i_2, n_1, n_2, *test*, *result* $\in$ s-expression

Here `my-do` must not be an ordinary function, because none of the 'arguments' like v_1, i_1, n_1, *test*, or *result* are to be evaluated before this rewrite rule is applied. Rather, the rewrite rule is to be applied first without evaluating any arguments, and then the result of the rewrite rule is to be evaluated.

We will use two ideas to implement this rewrite rule: macros and backquotes. We will describe the latter first.

Backquotes solve the problem of how to easily compute:

```
(let ((loop-f
       #'(lambda (lf v1 v2)
           (cond (test result)
                 (t (funcall lf lf n1 n2))))))
     (funcall loop-f loop-f i1 i2)))
```

given s-expression values for v_1, v_2, i_1, i_2, n_1, n_2, *test*, and *result*. Using backquote the computation is:

```
`(let ((loop-f
        #'(lambda (lf ,v1 ,v2)
            (cond (,test ,result)
                  (t (funcall lf lf ,n1 ,n2))))))
      (funcall loop-f loop-f ,i1 ,i2)))
```

where `v1` is a COMMONLISP variable whose value is the s-expression to be substituted wherever '`,v1`' appears inside the backquoted s-expression, and ditto for `v2`, `i1`, `i2`, `n1`, `n2`, `test`, and `result`.

So the idea is that in a backquoted expression, the comma becomes an operator, such that for an s-expression β, '`,`β' means evaluate β and substitute the result into the backquoted expression in place of '`,`β'.

Some thought shows that backquote and comma can be implemented by the following input equivalence rules:

$$
\begin{array}{lcl}
\texttt{`}\alpha & \equiv & \texttt{'}\alpha \\
\texttt{`(}\beta\ \texttt{.}\ \gamma\texttt{)} & \equiv & \texttt{(cons `}\beta\ \texttt{`}\gamma\texttt{)} \\
\texttt{`(,}\beta\ \texttt{.}\ \gamma\texttt{)} & \equiv & \texttt{(cons }\beta\ \texttt{`}\gamma\texttt{)} \\
\texttt{`(}\beta\ \texttt{.}\ \texttt{,}\gamma\texttt{)} & \equiv & \texttt{(cons `}\beta\ \gamma\texttt{)} \\
\texttt{`(,}\beta\ \texttt{.}\ \texttt{,}\gamma\texttt{)} & \equiv & \texttt{(cons }\beta\ \gamma\texttt{)}
\end{array}
$$

where $\alpha \in$ atom, $\beta, \gamma \in$ s-expression-extended-with-commas

Here β and γ are s-expressions that may contain commas.

By way of an example, these rules give:

```
  `(+ ,(+ 5 2) ,(- 5 2))
≡ `(+ . (,(+ 5 2) ,(- 5 2)))
≡ (cons `+ `(,(+ 5 2) ,(- 5 2)))
≡ (cons '+ `(,(+ 5 2) ,(- 5 2)))
≡ (cons '+ `(,(+ 5 2) . (,(- 5 2))))
≡ (cons '+ (cons (+ 5 2) `(,(- 5 2))))
≡ (cons '+ (cons (+ 5 2) `(,(- 5 2) . nil)))
≡ (cons '+ (cons (+ 5 2) (cons (- 5 2) `nil)))
≡ (cons '+ (cons (+ 5 2) (cons (- 5 2) 'nil)))
```

Figure 1.28 gives examples of backquoted expressions input to a LISP interpreter. Many LISP interpreters, including the one used in the figure, adjust the above input rules so that each backquoted s-expression becomes what these rules say but with extra information attached. The extra information is typically used to print the backquoted s-expression in a manner similar to the way it was input.

In the case of the interpreter in the figure, the extra information is encoded by encapsulating the result as we have defined it as the third element of a list whose other elements describe how to print the s-expression. The `whatis` function in the figure extracts the meaningful third element of this list.

```
;; Backquote notation:

;; In CLISP `... evaluates to a list of the
;; form:
;;
;;      (system::backquote ppp vvv)
;;
;; where vvv is what one would expect from `
;; and ppp is an aid to printing the expression
;; properly.  So to see vvv we define:
;;
(defun whatis (s-expression) (caddr s-expression))
--->
WHATIS
```

```
(whatis '`x)
--->
'X

(whatis '`(x . y))
--->
'(X . Y)

(whatis '`(,x . y))
--->
(CONS X 'Y)

(whatis '`(x . ,y))
--->
(CONS 'X Y)

(whatis '`(,x . ,y))
--->
(CONS X Y)
```

```
(whatis '`(b c d ,e))
--->
(LIST 'B 'C 'D E)

(whatis '`(+ ,(+ 5 2) ,(- 5 2)))
--->
(LIST '+ (+ 5 2) (- 5 2))

`(+ ,(+ 5 2) ,(- 5 2))
--->
(+ 7 3)
```

Figure 1.28: Backquote (`) Notation

```
(defmacro my-do
  (((v1 i1 n1)          ; (my-do ((v1 i1 n1)
    (v2 i2 n2))         ;         (v2 i2 n2))
   (test result))   ;        (test result))

  `(let ((loop-f
          #'(lambda (lf ,v1 ,v2)
              (cond (,test ,result)
                    (t (funcall lf lf ,n1 ,n2))))))
        (funcall loop-f loop-f ,i1 ,i2)))
--->
MY-DO

(macroexpand '(my-do ((i 1 (+ i 1))
                      (sum 0 (+ sum i)))
                     ((> i 100) sum)))
--->
(LET
  ((LOOP-F
      #'(LAMBDA (LF I SUM)
          (COND ((> I 100) SUM)
            (T (FUNCALL LF LF (+ I 1) (+ SUM I)))
  ))    ) )
  (FUNCALL LOOP-F LOOP-F 1 0)
)

(my-do ((i 1 (+ i 1))
        (sum 0 (+ sum i)))
  ((> i 100) sum))
--->
5050

(sum 1 100)
--->
5050
```

Figure 1.29: Example `Do` Macro

Also this particular LISP interpreter often simplifies the results by using a single COMMONLISP '`list`' function call in place of a nested set of '`cons`' function calls.[24]

So now we know how to compute the result of the rewrite, the next question is how to get the arguments defined, and get the computation of the rewrite right side to occur. Consider a typical call on `my-do`:

```
(my-do ((i 1 (+ i 1))
        (sum 0 (+ sum i)))
  ((> i 100) sum))
```

Close inspection shows that there are exactly two arguments in this call:

```
((i 1 (+ i 1))
 (sum 0 (+ sum i)))        and   ((> i 100) sum))
```

If we match these two arguments up with the patterns:

```
((v1 i1 n1)
 (v2 i2 n2))       and   (test result))
```

and assign to the variables in the pattern the corresponding s-expressions in the arguments, we get the assignments we want:

```
v1      ↦  i
i1      ↦  1
n1      ↦  (+ i 1)
v2      ↦  sum
i2      ↦  0
n2      ↦  (+ sum i)
test    ↦  (> i 100)
result  ↦  sum
```

This is part of what the following '**macro definition**' does:

```
(defmacro my-do
  (((v1 i1 n1)
    (v2 i2 n2))
   (test result))
  ...)
```

[24] `(list` β_1 β_2 `...` β_n`)` $\Longrightarrow$ `(cons` β_1 `(cons` β_2 `(cons ...(cons` β_n `nil)...)))` and `(list*` β_1 β_2 `...`β_{n-1} β_n`)` $\Longrightarrow$ `(cons` β_1 `(cons` β_2 `(cons ...(cons` β_{n-1} β_n`)...))`

It matches up the unevaluated arguments in a call to `my-do` to the patterns of variables in the macro definition of `my-do` and assigns variables accordingly.

The other thing a macro call does is after the arguments are assigned, it evaluates the body of the macro definition twice. The first time it gets a piece of code which when evaluated a second time gives the final result.

Thus when a function defined by `defun` or a closure expression is called, the arguments are evaluated first, the function rewrite is then applied, and the result is ready for further evaluation. In contrast, when a **macro** defined by `defmacro` is called, the arguments are not evaluated, the macro rewrite is then applied, and the result is evaluated a first time to get the right side of the rewrite, which is called the **expansion** of the macro call. This expansion is then ready for further evaluation.

Figure 1.29 runs the `my-do` `defmacro` that we have been discussing through the interpreter. The `macroexpand` function is used in this figure to return the expansion of a macro call, so the action of the macro can be seen. When a macro is called normally, this expansion is subject to one further evaluation to get a final result.

Exercise 1.7.2.1 Find an s-expression that contains no backquotes and is equivalent on input (≡) to the following:

(a)	`` `(,x ,y ,z) ``	(d)	`` `(let (,v ,e) (+ ,v 1)) ``
(b)	`` `(x (,y . z)) ``	(e)	`` `(cond (,test 0) (t 1)) ``
(c)	`` `(+ 7 .  ,r) ``	(f)	``` `',x ```

Exercise 1.7.2.2 Give rewrite rule definitions (not `defmacro`'s) for the following macros, using the basic list manipulation and data predicate functions we have described in this chapter:

(a) A macro `my-or` that takes two unevaluated s-expressions, evaluates the first, and if the value is non-`nil`, returns that value without evaluating the second s-expression. But if the value is `nil`, the second s-expression is evaluated and its value is returned. Use the COMMONLISP '`cond`' special function in the definition. Example uses:

```
(my-or (oddp 4) (oddp 9)) ===> t
(my-or (oddp 4) (oddp 8)) ===> nil
(my-or (oddp 4) 9) ===> 9
(my-or 8 9) ===> 8
(my-or 8 (car 9)) ===> 8
```

(b) A macro `my-when` that evaluates a first s-expression, and if the value is not `nil`, evaluates a second s-expression and returns its value. If the value of the first s-expression is `nil`, `nil` is returned and the second s-expression is ignored. Use the COMMONLISP '`cond`' special function in the definition. Example uses:

```
(my-when (oddp 7) 9) ===> 9
(my-when (oddp 8) 9) ===> nil
(my-when (oddp 8) (car 9)) ===> nil
```

(c) A macro `my-let` that does the same thing as COMMONLISP '`let`' when presented with a call of the form:

```
(let ((v1 e1)
      (v2 e2))
  e)
```

You must pattern your definition after the definition of `my-do` in the text, by defining a closure and using a `funcall`. You may not use any COMMONLISP macro or special function in the rewrite (you do <u>not</u> need to use '`let`' in the rewrite).

Exercise 1.7.2.3 Give the macro definitions (i.e., the `defmacros`'s) for the macros of the last exercise, using the basic list manipulation and data predicate functions we have described in this chapter:

(a) `my-or` (b) `my-when` (c) `my-let`

1.7.3 Tail Recursion

The **depth** of a tree is the number of levels in the tree. It is also the length of the longest path from the root of the tree to any leaf, where the path must start at the root and follow arrows until it gets to the leaf.

For a call tree, the depth is a measure of the number of functions that may be executing at one time. Normally each function execution takes some memory space, so the depth of the call tree is a measure of the maximum memory space required to execute the function call at the root of the tree.

One problem with recursive programming is that the call tree depth of a recursive function may be very large, and the computer may run out of space. The memory space used by function executions is called the **stack**, and so the problem is that the computer may run out of stack space, stop the program, and and give an 'out of stack space' error message.

However, there is an optimization that is used by all modern compilers that can eliminate running out of stack space in important cases.

A **tail recursive** function call is a call by a function to itself done as the last step in the calling function.

The optimization is to replace a tail recursive call by code that reuses the memory of the calling function execution for the called function execution. Since the calling function is not going to do anything else, its memory is no longer needed. Since the called function is the same as the calling function, the amount of memory the called function execution needs exactly matches the amount used by the calling function, making reuse especially easy.

A recursive function is called **tail recursive** if its only recursive calls are tail recursive calls to itself. The optimization just described is called the **tail recursive optimization**.

The function call `(random 2)` randomly returns either a `0` or a `1` with equal probability. Suppose we want to make this call n times and sum up all the results, to see if the sum is approximately $n/2$. We could do this with the function defined by the rewrite rules:

$$
\begin{array}{l}
\texttt{(random-sum 0)} \Longrightarrow \texttt{0} \\
\texttt{(random-sum } n\texttt{)} \Longrightarrow \texttt{(+ (random 2)} \\
\qquad\qquad\qquad\qquad\quad \texttt{(random-sum (- } n \texttt{ 1)))} \\
\qquad\qquad\qquad\qquad\qquad \text{if } \texttt{(> n 0)} \\
\text{where } n \in \text{integer}
\end{array}
$$

However, this function is <u>not</u> tail recursive, as the last thing it calls is `+`, and not itself. Therefore if we called the function with n equal to one hundred million, we might need memory space for one hundred million <u>simultaneous</u> function executions.

But our function can be rewritten to be tail recursive by switching to the following rules:

(`tail-recursive-random-sum` n)
$\Longrightarrow$ (`tail-recursive-random-sum` n `0`)
(`tail-recursive-random-sum 0` $result$) $\Longrightarrow$ $result$
(`tail-recursive-random-sum` n $result$)
$\Longrightarrow$ (`tail-recursive-random-sum` (`-` n `1`)
(`+ (random 2)` $result$))
if `(> n 0)`
where n, $result \in$ integer

This function will be optimized by the compiler, and not have the space problems of the previous function.

Figure 1.30 defines both the functions just discussed and Figure 1.31 shows their execution. The last thing in Figure 1.31 is a call to the non-tail-recursive function that does in fact cause the interpreter to run out of stack space and 'blow up'. There is no error message in the figure because none appears in the output file of the run, although an error message does appear on the user's terminal. This particular LISP interpreter does not recover gracefully from running out of stack space.

Some trickery is required in presenting these functions to the interpreter, because some LISP compilers are smart enough to rewrite the non-tail-recursive `random-sum` function into a tail recursive equivalent, based on their knowledge of the `+` operator. The trickery is to hide the `+` operator inside the undefined `my-add` function until after the `random-sum` and `tail-recursive-random-sum` functions have been compiled.

Any algorithm that can be programmed using an iterative construct such as a '`do`' statement can also be programmed without iteration by using a tail recursive function. Without the tail recursive optimization, however, algorithms that could be done by iteration would require very much less memory space to execute iteratively than they would to execute recursively. The tail recursive optimization converts a tail recursive call to code that copies new values into the places occupied by the current arguments and then jumps back to the beginning of the current function. Thus the optimization literally constructs an iterative program out of a recursive one.

```
;; Tail Recursiveness

;; (my-add m n) ===> (+ m n) is defined below

;; (random-sum 0) ===> 0
;; (random-sum n) ===> (+ (random 2)
;;                        (random-sum (- n 1)))
;;                     if (> n 0)
;;
;; n is an integer
;;
(defun random-sum (n)
  (cond ((= n 0) 0)
        ((> n 0) (my-add (random 2)
                         (random-sum (- n 1))))))
--->
RANDOM-SUM

;; (tail-recursive-random-sum n)
;;     ===> (tail-recursive-random-sum n 0)
;; (tail-recursive-random-sum 0 result) ===> result
;; (tail-recursive-random-sum n result)
;;     ===> (tail-recursive-random-sum (- n 1)
;;              (+ (random 2) result))
;;          if (> n 0)
;;
;; n, result are integers
;;
(defun tail-recursive-random-sum
          (n &optional (result 0))
  (cond ((= n 0) result)
        ((> n 0)
         (tail-recursive-random-sum
          (- n 1) (my-add (random 2) result)))))
--->
TAIL-RECURSIVE-RANDOM-SUM
```

Figure 1.30: Tail Recursion Example: Page 1 of 2

```
;; Compile the above BEFORE MY-ADD is defined, so
;; compiler cannot perform any fancy optimization to
;; make RANDOM-SUM tail recursive when it is not
;; directly so.

(proclaim '(optimize (compilation-speed 0)
                     (debug 1) (speed 3) (safety 3)))
--->
NIL

(compile 'tail-recursive-random-sum)
--->
TAIL-RECURSIVE-RANDOM-SUM

(compile 'random-sum)
--->
RANDOM-SUM

(defun my-add (x y) (+ x y))
--->
MY-ADD

(compile 'my-add)
--->
MY-ADD

(random-sum 1000)
--->
473

(tail-recursive-random-sum 1000)
--->
525

(tail-recursive-random-sum 1000000)
--->
500157


;; The following must be the last thing in the file
;; as it causes an unrecoverable error in some
;; COMMONLISP systems.

(random-sum 1000000)
--->
```

Figure 1.31: Tail Recursion Example: Page 2 of 2

Exercise 1.7.3.1 Which of the following function definitions are tail-recursive?

(a) The `power` function of Figure 1.8.
(b) The `fast-power` function of Figure 1.9.
(c) The Fibonacci function `f` of Exercise 1.4.7.1.
(d) The `my-nth` function of Figure 1.12.
(e) The `my-append` function of Figure 1.13.
(f) The `my-reverse` function of Figure 1.14.
(g) The `my-member` function of Figure 1.16.
(h) The `my-equal` function of Figure 1.17.

Exercise 1.7.3.2 Give rewrite rule definitions (<u>not</u> `defun`'s) for tail-recursive, non-iterative, versions of the following functions:

(a) The `power` function of Figure 1.8.
(b) The Fibonacci function `f` of Exercise 1.4.7.1.

Exercise 1.7.3.3 Give the function definitions (i.e., `defun`'s) for tail-recursive, non-iterative, versions of the following functions:

(a) The `power` function of Figure 1.8.
(b) The Fibonacci function `f` of Exercise 1.4.7.1.

Exercise 1.7.3.4

(a) Why is it hard to give a tail-recursive function definition of the `my-append` function of Figure 1.13, while it is easy to give one for the `sum` function of Figure 1.22?
(b) Suggest a way of writing `my-append` using tail-recursion and an additional function.

1.8 Grammars

In this section we will firm up our understanding of grammars and introduce grammar related terminology.

A **grammar** consists of a set of **grammar equations** that define **units**. A grammar has a **domain** that is a set, and a unit is a subset of this domain that has a **unit name**. The equation that defines a unit named X has the form:

$$\text{X} ::= \ldots$$

which is read 'X is ...'. The '::=' is a fancy kind of equals sign that indicates this is a grammar equation defining a unit.[25]

Different grammars have different domains, but usually the domain of a grammar is the set of all sequences of zero or more somethings. For example, the sequences of zero or more characters. In computer science the word '**string**' is often used to mean sequence, so a 'character string' is just a sequence of characters.[26]

A **constant** in a grammar equation is a symbol that represents some element of the grammar's domain. Inside the grammar equation, this symbol represents the set which has the represented domain element as the one and only element of the set. Thus if the domain is the set of character strings, and '0' represents the character string consisting of the single digit 0, then inside a grammar equation '0' represents the set {0}, and the grammar equation:

$$\mathrm{X} ::= \mathtt{0}$$

says that the unit named X is the set {0} whose only member is 0.

The symbol '|' is used to denote set union, often read as 'or' when used in grammar equations. Thus the equation:

$$\mathrm{digit} ::= \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \mathtt{3} \mid \mathtt{4} \mid \mathtt{5} \mid \mathtt{6} \mid \mathtt{7} \mid \mathtt{8} \mid \mathtt{9}$$

says that 'digit' names a unit that is the set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

The most commonly used grammar equation operator is 'concatenation', read as 'is followed by', and this operator has no symbol of its own, but is implied by writing two grammar equation expressions next to each other. Thus

$$\mathrm{Y} ::= \mathtt{0}\ \mathrm{digit}$$

defines a unit Y that is a set of the 10 members '00', '01', '02', ..., '09', and

$$\mathrm{Z} ::= \mathrm{digit}\ \mathrm{digit}$$

[25]Grammar equations are also called **production rules**, and grammar units are also called **non-terminals**.

[26]Grammars can also be written to define sets of trees.

defines a unit Z that is a set of the 100 members '`00`', '`01`', '`02`', . . . , '`10`', '`11`', '`12`', . . . , '`97`', '`98`', '`99`'.

In general the unit X followed by the unit Y is the set just containing each concatenation of a member of X with a member of Y.

Constants such as '`0`' in the definition of 'digit' are also called **terminals**, because they have no defining equation, and terminate the definition process, even though they have an implicit definition. The names of units are sometimes called **non-terminals**.

If the domain of a grammar is the set of character strings, some name must be given to the string that contains zero characters. This string is called the '**empty string**' and is often given the name 'ϵ' or '**empty**'.

In this section we give examples that assume the domain of the grammar is the set of all finite character strings, and use the name 'empty' for the empty string. In the rest of the book most of the grammars have as their domain a set of lexeme strings, where lexemes are numbers, symbols, and punctuation marks. Lexemes will be discussed in more detail below.

The four equations:

X ::= 0
X ::= empty 0
X ::= 0 empty
X ::= empty 0 empty

all define the same set. The most common use of empty is in equations like:

optional-X ::= empty | X
X-sequence ::= empty | X X-sequence

We just defined X-sequence to be any sequence of zero or more X's. This definition is recursive: X-sequence is defined in terms of itself. Viewed as a definition of a set, what can it mean?

It means that X-sequence is the smallest set satisfying the equation. This is just a convention one always uses when defining grammars. Thus an equivalent equation would be:

X-sequence ::= empty | X | X X | X X X | X X X X | . . .

except in this equation one must imagine that the '...' stands for longer and longer sequences of X's separated by |'s, so the equation is infinite and depends upon correct interpretation of '...' by the reader. The recursive equation is precise.

For the equation:

$$\texttt{0}\text{-sequence} ::= \text{empty} \mid \texttt{0}\ \texttt{0}\text{-sequence}$$

there are no sets of finite character strings satisfying the equation other than the smallest set, as one can prove by recursion on n that the first n things in any `0`-sequence must be `0`'s. However, for the equation:

$$\text{Z} ::= \text{empty} \mid 0 \mid \text{Z Z}$$

the set of all sequences of zero or more digits satisfies the equation, but is not the smallest set doing so.

How does one know that there are smallest sets satisfying some set of grammar equations? Consider the following 'proof' that Z contains '0 0':

$$\begin{array}{lcl} \text{Z} & \Longrightarrow & \text{Z Z} \\ & \Longrightarrow & 0\ \text{Z} \\ & \Longrightarrow & 0\ 0 \end{array}$$

At each step, one of the non-terminals is replaced by one of the | separated grammar subexpressions from the right side of the equation defining the non-terminal. Such a proof that a sequence of terminals is in a unit is called a **production**. The set of all sequences of terminals that can be produced from a non-terminal is the smallest set that satisfies the equation defining the non-terminal.

Sequences are so common that they get their own operator, '$\star$', as in:

$$\text{X-sequence} ::= \text{X}^{\star}$$

which defines the same set as the above definitions of X-sequence. Thus '$\star$' means 'zero or more'.

The operator '$^+$' is used to mean 'one or more', making the following two definitions equivalent:

non-empty-X-sequence ::= X | X X-sequence
non-empty-X-sequence ::= X$^+$

Curly brackets '{ }' are used in grammar equations just as parentheses are used in arithmetic expressions. Thus:

non-empty-X-list ::= X { , X }*

is equivalent to:

non-empty-X-list ::= X Xrest*
Xrest ::= , X

Sometimes there is confusion between grammar operator symbols and terminals. To remove such confusion, terminals may be quoted. In the equation:

curly-X-list ::= '{' '}' | '{' X { , X }* '}'

the quoted curly brackets are terminals: '{' denotes the set whose only member is {; and the unquoted curly brackets delimit the grammar subexpression ', X'.

Similarly there may be confusion between unit names and terminal constants, which may be resolved by quoting any terminal constant that looks like a unit name.

Many other notations may supplement those just given for grammars. We will give three examples to indicate the variety of what is possible.

The first example is the use of English inside angle brackets < > to describe a subset of the domain, as in:

graphic-character ::= <any ASCII character that makes a mark>

The second example is the use of 'X except Y' to denote the set of all elements of X that are not in Y. Thus

octal-digit ::= digit except { 8 | 9 }

defines an octal digit to be any digit except '8' or '9'.

The third example is the convention that unit names of the form 'optional-X' are implicitly defined to mean:

$$\text{optional-X} ::= \text{empty} \mid \text{X}$$

That is, as long as X is defined, no explicit definition need be given for 'optional-X', which is assumed to be defined by the equation just given. Names with various other forms, such as 'X-sequence' and 'non-empty-X-sequence', can be defined by similar conventions.

The grammar of Figure 1.32 defines the '**lexemes**' for the subset of LISP we have been using. A 'lexeme' is just a number, a symbol, or a punctuation mark. The grammars we have given before this section, and almost all other grammars in this book, have domains consisting of lexeme strings: i.e., sequences of zero or more lexemes.

To simplify the grammar of Figure 1.32 we have assumed all numbers are integers, and none are ratios or floating point.

The grammar in Figure 1.32 is '**ambiguous**' in many ways. The file '`x1`' could be interpreted as the two lexemes '`x`' and `1`', or as the single lexeme '`x1`', according to the grammar equations. Some extra rule is needed to disambiguate this case. The extra rule, given in the figure, is that at any point in scanning the file the longest lexeme is preferred. Thus '`x1`' is the only lexeme in the file '`x1`'.

Grammars given elsewhere in this book are not ambiguous unless specifically noted.

It is often a good idea to use notes to explain points about a grammar, such as how to resolve ambiguities. The grammar in Figure 1.32 has notes which begin with the **bullet** character '•'.

Exercise 1.8.1 For each of the following one equation grammars, what is the set of finite character strings named by X?

(a) $X ::= \texttt{a} \mid \texttt{b}\ X$
(b) $X ::= \texttt{a}^{+} \mid X\ \texttt{b}\ X$
(c) $X ::= \texttt{a}\ \texttt{b} \mid \texttt{a}\ \texttt{b}^{\star}\ X$

file ::= whitespace { lexeme whitespace }*

lexeme ::= mark | number | symbol

whitespace ::= whitespace-item*
whitespace-item ::= whitespace-character | comment | non-legal-character
whitespace-character ::= SPACE | TAB | LINE-FEED | CARRIAGE-RETURN
comment ::= ; non-end-of-line-character*
end-of-line-character ::= LINE-FEED | CARRIAGE-RETURN | FORM-FEED

mark ::= `.` | `(` | `)`

number ::= { `+` | `-` | empty } digit$^+$

symbol ::= symbol-character$^+$
symbol-character ::= letter | digit | `+` | `-` | `*` | `/` | `=` | `<` | `>` | `~` | `@` | `$` | `%` | `^` | `&` | `_` | `.` | `?` | `!` | `[` | `]` | `{` | `}`

legal-character ::= symbol-character | `(` | `)` | whitespace-character | `;`

- The domain of the grammar is the set of all ASCII character strings.
- At each point in reading the file, the longest legal whitespace-item or lexeme is preferred.
- A number is preferred over a symbol.
- The '`.`' mark is preferred over a symbol.

Figure 1.32: Lexical Grammar for a Subset of LISP

Exercise 1.8.2 Write grammars in which X names the following sets of finite character strings:

(a) All non-empty strings of `a`'s and `b`'s.
(b) All strings of `a`'s and `b`'s containing at least 2 `b`'s.
(c) All non-empty strings of alternating `a`'s and `b`'s.
(d) All non-empty strings of `a`'s and `b`'s with an even number of `a`'s.
(e) All non-empty strings of `a`'s and `b`'s that are palindromes (i.e., which are not changed by reversing the order of characters in the string, e.g., `aba`, `abba`, and `aabaabaa`).

1.9 Labeled Graphs and Trees

In this section we will firm up our understanding of labeled graphs and trees and introduce related terminology.

1.9.1 Labeled Graphs

A **labeled graph** consists of a set of **nodes** and a set of **arcs**, each arc connecting a pair of nodes, with optional **labels** on each node and each arc. See Figure 1.18 for an example. Labels can be text, numbers, or other things.

An arc may be **directed**, as an arrow, or **undirected**, as a line. A directed arc has a **source** node and a **destination** node (the source node is also called the **origin** and the destination node the **target**). An undirected arc simply connects two nodes. Two nodes connected by an undirected arc are called **neighbors** and are said to be **adjacent**.

Applied mathematicians use the word **vertex** instead of node, and the word **edge** instead of arc. Software engineers are more likely to use the words node and arc. Also, unlike mathematicians, the graphs of software engineers may have several distinct arcs with the same label and the same source, or even with the same label, same source, and same destination.

Arcs are sometimes called **connections**. Directed arcs are sometimes called **arrows** (as we usually do elsewhere in this book). A graph all of whose arcs are directed is called a **directed graph**. A graph all of whose arcs are undirected is called an **undirected graph**.

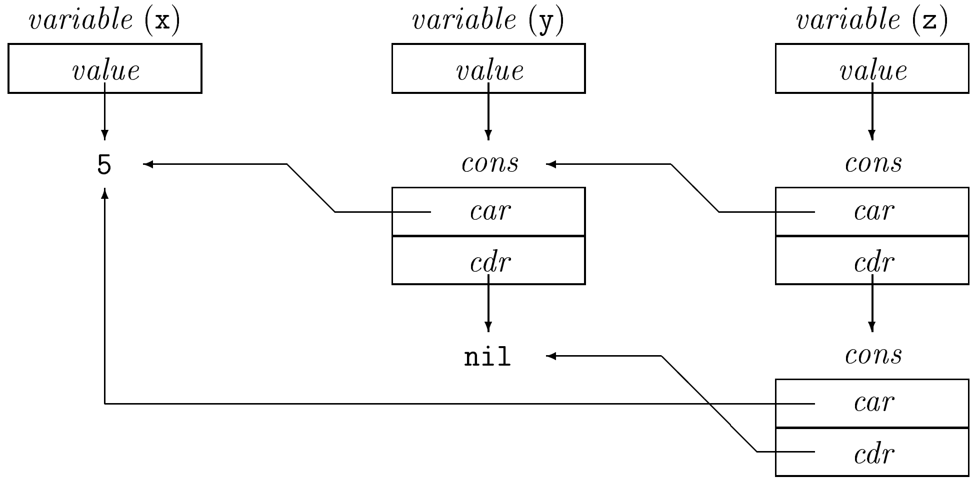


Figure 1.33: Basic LISP Example Data Structure: Alternative Depiction

Memory contents are often described as a set of nodes called **object**s connected by directed arcs called **pointers**. These pointers are often directly implemented by hardware addresses, i.e. by C language pointers. To implement an undirected arc, the undirected arc is often replaced by two directed arcs, one in each direction, with each directed arc being implemented by a hardware address.

There are many ways of depicting a labeled graph. The graph of Figure 1.18 could be represented alternatively as in Figure 1.33, for example.

Exercise 1.9.1.1 Draw the following graphs:

(a) A graph with nodes labeled 1, 2, 3, 4, and 5 and an unlabeled arrow from node m to node n whenever $m < n$.

(b) A graph with nodes labeled 1, 2, 3, 4, and 5 and an arrow labeled 3 from node m to node n whenever $m = n \bmod 3$ and $m > n$.

(c) A graph with nodes labeled `ab`, `abba`, `bb`, `babb`, and `babba` and an unlabeled arrow from one node to another if the label of the source node is a substring of the label of the destination node. Thus there would be an arrow from `bb` to `babb`.

1.9.2 Trees

A non-empty **path** through a directed graph is a non-empty sequence of arrows such that the destination of each arrow but the last is the source of the next arrow. The source of the first arrow is the **source** of the path; the destination of the last arrow is the **destination** of the path.

An **empty path** is just a node which is the both source and destination of the path: there are no arrows in an empty path.

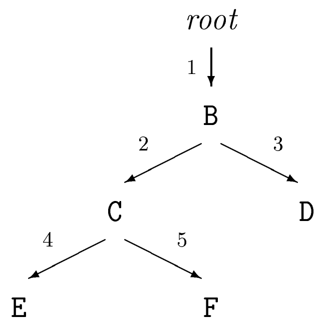
If P is a path with source X and destination Y, we say that P is a path **from** X **to** Y.

A node Y of a directed graph can be **reached** from a node X if there is a path from X to Y.

A directed graph is called a **tree** if and only if:

1. There is a unique node called the **root** that is not the destination of any arrow.
2. Every non-root node is the destination of exactly one arrow.
3. Every node can be reached from the root node.

A simple example is:



Computer scientists usually draw trees upside down.

If there is an arrow from a node X to a node Y in a tree, X is called the **parent** of Y and Y is called the **child** of X. Thus in the above example, `C` is the parent of `E` and `F`, and `C` is the child of `B`.

A tree node that is not the parent of any other node, i.e. that is not the source of any arrow, is called a **leaf**. In the above the leaves are D, E, and F.

Examples of trees are given in Figure 1.7. In these trees, the arrows originating at any node are labeled 1, 2, 3, ... Whenever arrows are labeled in this way, the labels may be omitted when the arrows are written from left to right in order of increasing labels. Thus the arrow labels are usually omitted in the trees given in Figure 1.7. The omitted labels may be reinserted by simply numbering all arrows with a given origin 1, 2, 3, ..., from left to right.

Exercise 1.9.2.1 For each of the following arithmetic expressions draw a tree like those at the beginning of this chapter with a node for each operator, variable, and number such that the children of an operator node are trees of the operand subexpressions of that operator in the original expression. Label the nodes with their operator, variable, or number and the arrows with the number of their operand (1, 2, 3, ...).

(a) `(- (* x y))`
(b) `(+ (+ 1 2) (+ 3 4))`
(c) `(+ 8 (* 2 x) (* 3 x x))`
(d) `(+ 1 x (- y) (- (* y y))))`

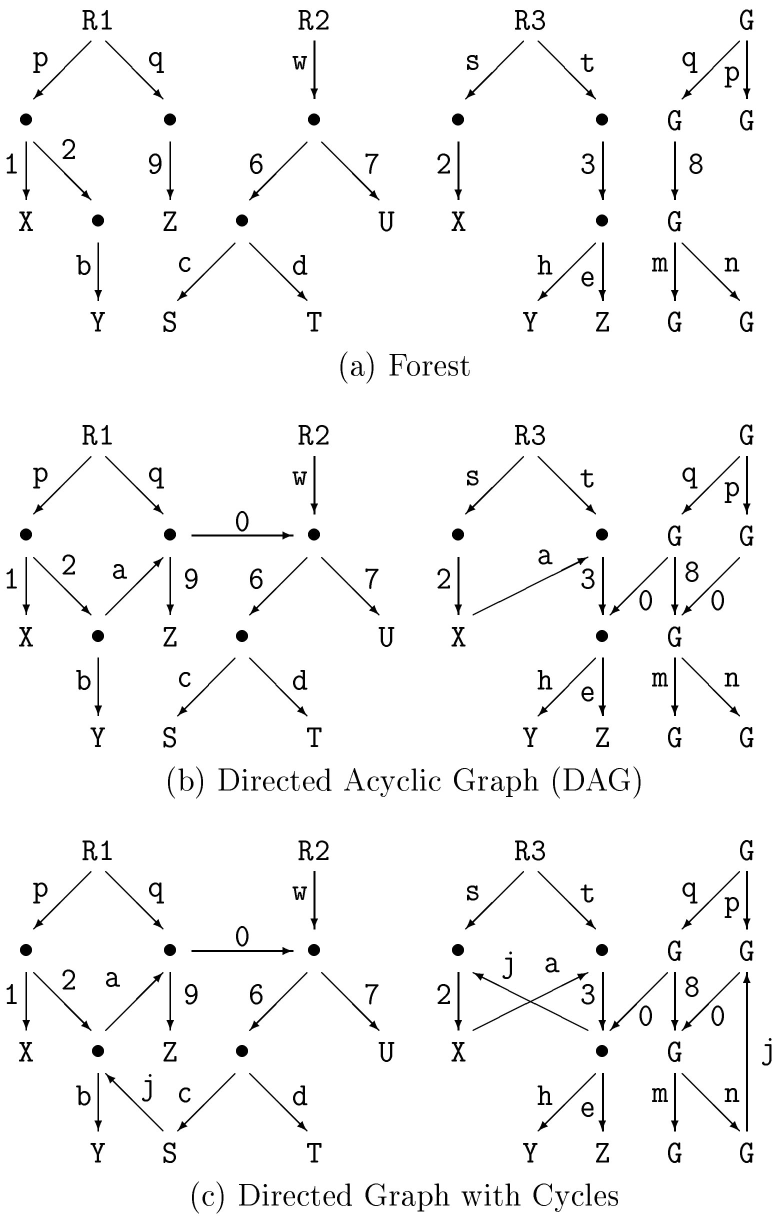
1.9.3 Storing Information in Directed Graphs

The reader may want to skim this section and come back to it later for reference purposes.

Directed graphs with labeled arrows are often used to store information. Some of the nodes of the graph are designated as places to start searching for information: these nodes are called **roots**. Information is found by starting at a root and moving along a path to some node that holds the information.

For example, consider the graphs of Figure 1.34. In these graphs the roots are labeled R1, R2, or R3 (these roots are a matter of choice, and by choice nodes labeled G are not roots). The node with label T in each of these graphs is reached by following the path R2.w.6.d from root R2. Here the path is a sequence of three arrows, which have labels w, 6, and d.

In general we can describe any path in a directed graph by giving the source node of the path and the labels of the arrows. For this to work, each arrow must have a label, and the labels of all arrows with the same source must be different. We can denote a path by writing down a label for the source node followed by the labels of



(a) Forest

(b) Directed Acyclic Graph (DAG)

(c) Directed Graph with Cycles

Figure 1.34: Examples of Directed Graphs

the arrows in the path, separating these labels by some character such as '`.`' or '`/`' or '`\`' to get path names such as `R2.w.6.d` or `R2/w/6/d` or `R2\w\6\d`. These rules are typically followed in a directed graph used to store information.

Any node that cannot be reached (by a path) from a root is called **garbage**, because it represents information that can no longer be accessed. The garbage nodes in the graphs of Figure 1.34 are labeled with `G`. Note that no root can be a garbage node: any root can be reached from itself by an empty path.

A **cycle** in an directed graph is a non-empty path whose destination is the same as its source. In Figure 1.34 there are no cycles in graphs (a) and (b), but (c) has cycles `S.j.a.0.6.c`, `X.a.3.j.2` and `G.0.n.j` where here `X` is the node reached by the path `R3.s.2` and `G` is the node reached by the path `G.p` from the top node labeled `G`.

A directed graph is called **acyclic** if it has <u>no</u> cycles. A **directed acyclic graph** is called a **DAG**. Most graphs used to store information are DAGs. In Figure 1.34 graphs (a) and (b) are DAGs, but (c) is not as it has cycles.

A node that is the destination of more than one arrow is said to be **shared**. A tree, according to the definition given above, is a graph with no shared nodes in which if you start at any node and follow the arrows backwards, you come to a unique node that is not the destination of any arrow, namely the root of the tree.

A **forest** is a set of trees. The trees are not connected to each other, as two trees cannot share nodes, since a tree has no shared nodes. A finite DAG with no shared nodes is necessarily a forest, because you cannot follow a backward path indefinitely since the graph is finite and has no cycles.

In Figure 1.34 graph (a) is a forest but (b) and (c) are not as the nodes that are destinations of arrows labeled `0`, `a`, and `j` are shared.

There are two different definitions of root in a forest used to store information. First, we have a graph with some arbitrary set of nodes, R, as roots in the sense we have been discussing for any graph that is used to store information. Second, there are the roots of the trees in the forest. For any such forest, any tree root that is not in R will be a garbage node, and any tree whose root is in R will consist of non-garbage nodes.

It is important to have algorithms that find garbage in a directed graph with roots. These algorithms are simpler for finite DAGs, and more complicated for directed graphs with cycles.

The **reference count** of a node in a directed graph is the number of arrows which have that node as a destination. If one thinks of arrows as pointers, then the reference count is the number of times a node is pointed at, or referenced. For a forest, nodes that are roots of trees have reference count 0, and other nodes have reference count 1. For any directed graph, a shared node is just a node with a reference count greater than 1.

For a finite DAG with roots, if there are garbage nodes, then at least one of them must have a zero reference count, and none of them may be roots. So one non-root node that has zero reference count may be removed. This process may then be iterated until there are no garbage nodes.

This algorithm is called **reference counting garbage collection**.

For a directed graph with roots that has cycles, this process will not work, as indicated by the example in Figure 1.34(c), in which after the top node `G` has been removed, and the node reached from the top by the path `G.q` has been removed, the remaining `G` nodes will all have reference counts greater than 0.

The algorithm used to find garbage in this case is a **marking algorithm**. A mark is placed on each root. Then if there is any unmarked node X that is the destination of an arrow whose source is marked, the node X is marked. This process is continued until no arrow has a marked source and an unmarked destination. At this point, the unmarked nodes are exactly the garbage nodes. The unmarked nodes may then be removed.

The complete algorithm to remove garbage from a directed graph with cycles is called **mark and sweep garbage collection**. The word sweep refers to the end part of the algorithm where the unmarked nodes are removed, or 'swept away'.

1.9.4 Hypergraphs

The reader may want to skip or skim this section and come back to it later when reading about resolution theorem proving in Chapter 3.

An arc in a graph connects two nodes. But in the real world there are often relations between more than two nodes, and to describe these we may need fancier kinds of arcs. The general term for a graph some of whose arcs involve more than two nodes is a **hypergraph**[27].

[27] If you look at a book on hypergraphs, you will find that applied mathematicians have settled on a somewhat more restricted definition than we just gave, but it seems to us that broadening their definition is better than inventing a new word.

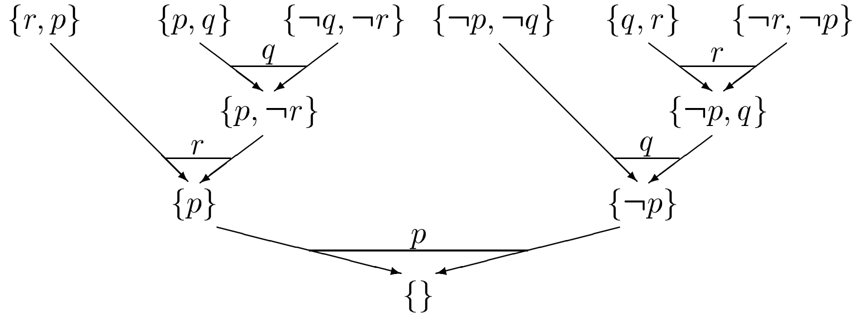


Figure 1.35: Example Resolution Theorem Proving Hypergraph

We briefly sketch one use of hypergraphs.

Many problems in discrete mathematics can be reduced to solving a set of boolean logic equations. One method for solving such equations is **resolution theorem proving**.

To give a simple minded example, consider the logic equation:

$$\begin{array}{rl}\mathbf{t} \equiv & (p \wedge q) \vee (q \wedge r) \vee (r \wedge p) \\ & \vee (\neg p \wedge \neg q) \vee (\neg q \wedge \neg r) \vee (\neg r \wedge \neg p)\end{array}$$

which says that given three true/false propositions, p, q, and r, either 2 are **t** or 2 are **f** (here p, q, and r are variables that can take the values **t** denoting 'true' or **f** denoting 'false'; $\wedge$ denotes boolean 'and'; $\vee$ denotes boolean 'or'; and $\neg$ denotes boolean 'not'). To solve this with resolution theorem proving we first negate both sides[28] to get:

$$\begin{array}{rl}\mathbf{f} \equiv & (\neg p \vee \neg q) \wedge (\neg q \vee \neg r) \wedge (\neg r \vee \neg p) \\ & \wedge (p \vee q) \wedge (q \vee r) \wedge (r \vee p)\end{array}$$

We then build the hypergraph in Figure 1.35 whose nodes are labeled by 'clauses'. Each clause is a set of 'literals', and each literal is a variable or the negation of a

[28]Using DeMorgan's Laws: see Chapter 2.

variable. The literals in a clause are implicitly connected together by $\vee$'s. The first clauses we consider are those from the last equation, and are written along the top of the hypergraph, but in a different order than they appear in the equation.

In this hypergraph our arcs are called **resolutions**. A resolution connects two source nodes to a single destination node. To make a resolution, the two source node labels must have a variable V in common, and the variable must appear negated in one of the source node labels, and non-negated in the other source node label. These two occurrences of V are said to 'cancel', and the destination label lists all the literals in either source node label, except these two canceling occurrences of V.

A resolution arc is labeled with the variable it cancels.

The last equation above turns out to be true if and only if there is a path in this hypergraph from the nodes in the equation to the node labeled by the empty set, $\{\}$. In this example, there is such a path.

Here we shall not attempt to properly explain resolution theorem proving or its serious uses. The important point for us is that the resolution algorithm involves a hypergraph with a non-standard definition of arc.

In this case, an arc is an arrow from an unordered pair of source nodes to a single destination node.

How do we properly define what a path is? The source of a path in this hypergraph is an unordered set of nodes, and the destination is a single node. An empty path is just a single node, which is the destination of the path, and any set containing that node, which is the source of the path.

A non-empty path from a source set S to a destination D is:

1. An arc whose source is $\{X, Y\}$ and whose destination is D.
2. A path from S to X.
3. A path from S to Y.

Thus we have defined paths recursively and precisely.

The reason it is important to use this hypergraph in discussing resolution theorem proving is that there are standard algorithms for searching ordinary graphs that can be modified in simple ways to search this kind of hypergraph for a path from some set, e.g. the set of clauses in an equation, to some destination node, e.g. $\{\}$.

In general, if you need to define a hypergraph, you need to precisely define the different kinds of arcs you want to use, and to adjust any definitions, such as that of path, accordingly.

Chapter 2

Symbolic Algebra

In this chapter we study how to perform algebraic calculations such as the one described at the beginning of Chapter 1:

```
5 + 2 * x - 3  ==>  2 * x + 5 - 3
               ==>  2 * x + (5 - 3)
               ==>  2 * x + 2
```

We will introduce the concept of a 'formal algebra' as a set of expressions and equational axioms, and we will study how to introduce rewrite rules that will rewrite every expression of a formal algebra into a 'normal form'. Sometimes this can be done so well that every equation rewrites to a normal form and two normal forms are equal expressions if and only if they are syntactically identical. Then the computer can decide whether two expressions are equal by rewriting both to normal form and comparing the normal forms syntactically.

A formal algebra that will be familiar to the reader is the algebra of rational arithmetic, which consists of expressions containing numbers, variables, and the four operators `+`, `-`, `*`, and `/`. We first study a simplified case of this algebra where only variables and the `+` operator are permitted in equations. Starting with this simplified case, we study how to determine whether two expressions are equal by rewriting each to a normal form. One goal of this chapter is to be sure the reader understands the notion of evaluation by rewriting.

Then we study Boolean algebra, which is the algebra of expressions containing the constants **t** (true) and **f** (false), variables, and the operators $\neg$ (not), $\wedge$ (and), and

$\vee$ (or). A second goal of this chapter is to be sure the reader understands Boolean algebra well.

A strong original motivation for developing LISP in the 1950's was to program computers to perform symbolic algebraic computations such as those in this chapter. The particular form of algebraic computation described in this chapter, computation by rewriting, is involved in many symbolic algorithms in computer science.[1]

2.1 The Algebra of Simple Expressions

We start with a simple example to illustrate the basic ideas.

Consider the formal algebra of variables and the + operator. The grammar of this algebra is:

$$\begin{array}{lcl} \text{simple-expression} & ::= & \text{variable} \\ & | & (\ \text{simple-expression} + \text{simple-expression}\ ) \\ \text{variable} & ::= & x \mid y \mid z \mid x_1 \mid y_1 \mid z_1 \mid \ldots \end{array}$$

Some examples of simple expressions are:

$$\begin{array}{lll} (x + y) & (x + (y + z)) & ((x + y) + z) \\ (x + (y + (z + (x_1 + x_3)))) & & ((x + x) + (x + x)) \end{array}$$

The equational axioms satisfied by this algebra are:

$$\begin{array}{ll} (x + y) = (y + x) & [\textbf{commutativity}] \\ ((x + y) + z) = (x + (y + z)) & [\textbf{associativity}] \end{array}$$

and an example computation using these axioms is:

$$\begin{array}{lll} & (z + (y + x)) & \\ = & (z + (x + y)) & \text{by commutativity} \\ = & ((z + x) + y) & \text{by associativity} \\ = & ((x + z) + y) & \text{by commutativity} \\ = & (x + (z + y)) & \text{by associativity} \\ = & (x + (y + z)) & \text{by commutativity} \end{array}$$

[1] See [Mit96], Chapter 3 in particular, for a more advanced treatment.

This computation serves to put the variables in lexically sorted (dictionary) order.

We call the formal algebra we have just described the **algebra of simple expressions**.

In general, a **formal algebra** is a set of **expressions** defined by a grammar, and a set of equations called **equational axioms**. The terminal symbols of the grammar may include symbols called **variables** that play a special role in the algebra, in that they obey rules such as 'equals can be substituted to produce equals', as is specified more precisely below. Each **equation** of the algebra has the form $e_1 = e_2$, where e_1 and e_2 are expressions of the formal algebra.

The algebra of simple expressions above has a grammar and two axioms: commutativity and associativity.

We will use $\equiv$ in this chapter to denote **syntactic identity**. So $e_1 \equiv e_2$ means that e_1 and e_2 consist of the same string of symbols. On the other hand, $e_1 = e_2$ is an equation of a formal algebra (which may or may not be 'true'). Thus for the algebra of simple expressions, `(`x `+` y`)` $\equiv$ `(`x `+` y`)`, while `(`x `+` y`) = (`y `+` x`)` is just an equation and not syntactic identity.

If e and e' are expressions and v is a variable of a formal algebra, then we write $[v \mapsto e']e$ for the expression made from e by substituting e' for <u>all</u> occurrences of v in e. Thus in the algebra of simple expressions:

$$[s \mapsto (x + y)](r + (s + s)) \equiv (r + ((x + y) + (x + y)))$$

Some equations of a formal grammar are provable. For example, the axioms are provable. One would expect that if $e_1' = e_2'$ is provable, e is any expression, and v is any variable, then $[v \mapsto e_1']e = [v \mapsto e_2']e$ should be provable. This is just a notational way of saying that if we substitute provably equal expressions e_1' and e_2' for some variable v in some expression e we get two provably equal expressions. We proceed to give a precise definition of provable equation.

An equation $e_1 = e_2$ is a **provable equation** of a formal algebra if and only if e_1 and e_2 are expressions of the algebra and at least one of the following is true:

1. $e_1 = e_2$ is an equational axiom of the algebra.

   Thus equational axioms are always provable. In the algebra of simple expressions, `(`x `+` y`) = (`y `+` x`)` is provable.

2. $e_1 \equiv e_2$.

   Thus syntactically identical expressions are always provably equal. In the algebra of simple expressions, `(`x` + `y`) = (`x` + `y`)` is provable.

3. $e_2 = e_1$ is a provable equation of the algebra.

   Thus the order of expressions in a provable equation can always be switched. In the algebra of simple expressions, since `(`x` + `y`) = (`y` + `x`)` is provable it follows that `(`y` + `x`) = (`x` + `y`)` is provable.

4. For some expression e, $e_1 = e$ and $e = e_2$ are both provable equations of the algebra.

   Thus if two expressions are both provably equal to a third expression e, the two expressions are provably equal. In the algebra of simple expressions, if `((`x` + `y`) + `z`) = (`x` + (`y` + `z`))` and `(`x` + (`y` + `z`)) = (`x` + (`z` + `y`))` are provable, then `((`x` + `y`) + `z`) = (`x` + (`z` + `y`))` is provable.

5. There are provable equations $e'_1 = e'_2$ and $e''_1 = e''_2$ and a variable v of the algebra such that $e_1 \equiv [v \mapsto e''_1]e'_1$ and $e_2 \equiv [v \mapsto e''_2]e'_2$.

   Thus if two provably equal expressions e''_1 and e''_2 are substituted for some variable v in two provably equal expressions e'_1 and e'_2 respectively, the two resulting expressions are provably equal. In the algebra of simple expressions, if `(`y` + `x`) = (`x` + `y`)` and `((`r` + `s`) + `t`) = (`r` + (`s` + `t`))` are provable, then substituting for the variable s we get that `((`r` + (`x` + `y`)) + `t`) = (`r` + ((`y` + `x`) + `t`))` is provable.[2]

In our algebra we can prove that `(`s` + (`r` + `s`)) = ((`s` + `s`) + `r`)`. While this is easy for a person to prove, it is hard for a computer to prove directly, because the computer has no efficient way to make the choices needed to construct a proof.

Exercise 2.1.1 Compute the following:

(a) `[`$x \mapsto$`(`z` + `z`)] (`x` + ((`x` + `x`) + `y`))`
(b) `[`$p \mapsto$`(`y` + `z`)] ((`p` + (`q` + `p`)) + `q`)`
(c) `[`$q \mapsto$`(`y` + `z`)] ((`p` + (`q` + `p`)) + `q`)`

[2]Variables in the formal algebra equations that we deal with in this chapter are all '**free variables**', meaning that any expression can be substituted for them in an equation. Free variables represent 'any value', in contrast to '**bound variables**' that represent a particular value, and that cannot be freely substituted for. For example, if you are asked to solve the equation '$x^2 + 2x + 3 = 0$' for x, then x is a bound variable.

Exercise 2.1.2 Prove the following, naming the axioms used at each step:

(a) $((x + y) + (z + w)) = (x + (y + (w + z)))$
(b) $(x + ((w + z) + x)) = ((x + x) + (z + w))$
(c) $(x + (y + (z + y))) = (((z + x) + y) + y)$

2.1.1 Rewriting to Normal Form

There is a general algorithm which a computer can use to prove $(s + (r + s)) = ((s + s) + r)$ efficiently. This algorithm replaces the equational axioms of the algebra by rewrite rules. For the algebra of simple expressions these rewrite rules are:

$(v_2 + v_1) \Longrightarrow (v_1 + v_2)$	if $v_1 << v_2$	[**commutativity 1**]
$(v_2 + (v_1 + e)) \Longrightarrow (v_1 + (v_2 + e))$	if $v_1 << v_2$	[**commutativity 2**]
$((e_1 + e_2) + e_3) \Longrightarrow (e_1 + (e_2 + e_3))$		[**associativity**]

where v_1, $v_2 \in$ variable; e, e_1, e_2, $e_3 \in$ simple-expression

Here we use $<<$ to denote the **lexical ordering** of variables: $v_1 << v_2$ means the variable v_1 occurs earlier in a dictionary ordering of variables than v_2. For example, $x << y$ but not $y << x$.

If we apply the above rewrite rules to the two sides of the equation $(s + (r + s)) = ((s + s) + r)$ we get:

$$
\begin{array}{lll}
 & (s + (r + s)) & \\
\Longrightarrow & (r + (s + s)) & \text{by commutativity 2} \\
 & ((s + s) + r) & \\
\Longrightarrow & (s + (s + r)) & \text{by associativity} \\
\Longrightarrow & (s + (r + s)) & \text{by commutativity 1} \\
\Longrightarrow & (r + (s + s)) & \text{by commutativity 2}
\end{array}
$$

We note that in both cases we end up with the same expression, $(r + (s + s))$, and that this expression cannot be rewritten further: no rewrite rule can be applied to it or any of its subexpressions. The algorithm is just this: to prove an equation, rewrite each side separately until no more rewriting is possible, and then check to see if both rewritten sides are syntactically identical.

As in Chapter 1, where we applied rewrite rules to evaluate LISP expressions, a rewrite rule may be applied to a whole expression of a formal algebra or to any of its subexpressions.

An expression to which no rewrite rule can be applied is said to be '**normal**'. Note that every subexpression of a normal expression is normal.

For the algebra of simple expressions, normal expressions must be **right associative**, meaning that the left operand of any `+` must be a single variable. If this were not the case, we could apply the associative rewrite rule to the expression. Also the variables in a normal expression must be sorted in non-decreasing lexical (dictionary) order. Otherwise, if two adjacent variables were in decreasing order in a right associative expression, we would be able to apply one of the two commutativity rewrite rules to the expression.

So in the algebra of simple expressions the normal expressions are just the right associative expressions with variables appearing in lexical order.

We could use any ordering of variables as long as given two distinct variables, one is before the other in the ordering. We also neglected to consider variable subscripts in defining variable order, but subscripts can be included in any way as long as every two distinct variables are ordered.

We have given an algorithm which we assert determines whether or not an equation is true. Several things have to be proved to show that this algorithm actually works.

First, we must show that the process of rewriting an expression terminates after a finite number of steps with a normal expression (an expression that cannot be rewritten further). This is true for the algebra of simple expressions, and therefore we say that the rewrite rules of this algebra are '**terminating**'.[3] We will prove this in Section 2.6.2.

To get an idea of how one proves that the rewrite rules of the algebra of simple expressions are terminating, consider just the commutativity rewrite rules. Count the number of pairs of variables in an expression that are in the wrong order: e.g., there are 2 such pairs in $(s + (t + r))$, namely s and r are one pair, and t and r are another pair. Then each application of a commutativity rewrite rule reduces the number of such wrongly-ordered variable pairs. Furthermore, an application of the associativity rewrite rule does not change the number of wrongly-ordered pairs. So at most a finite number of commutativity rewrite rules can be applied to rewrite any one expression, or otherwise the number of wrongly-ordered pairs in some expression would have to be negative, which is impossible.

Second, notice that there are sometimes two ways of rewriting an expression:

[3] Terminating algebras are also said to be '**strongly normal**'.

$$\begin{array}{lll} & \texttt{((}s\texttt{ + }r\texttt{) + }s\texttt{)} & \\ \Longrightarrow & \texttt{(}s\texttt{ + (}r\texttt{ + }s\texttt{))} & \text{by associativity} \\ \Longrightarrow & \texttt{(}r\texttt{ + (}s\texttt{ + }s\texttt{))} & \text{by commutativity 2} \\ & \texttt{((}s\texttt{ + }r\texttt{) + }s\texttt{)} & \\ \Longrightarrow & \texttt{((}r\texttt{ + }s\texttt{) + }s\texttt{)} & \text{by commutativity 1} \\ \Longrightarrow & \texttt{(}r\texttt{ + (}s\texttt{ + }s\texttt{))} & \text{by associativity} \end{array}$$

For the algebra of simple expressions it is true that no matter which way one chooses to rewrite an expression, the result in the end is the same. Therefore we say that the rewrite rules of this algebra are '**confluent**'.[4] We will prove this in Section 2.6.4. You can think of rewriting an expression as moving down stream in a river. If you have a choice of rewrite rules to apply at some point, it is like having a choice of branches in the river, and confluence assures us that we will come out at the same place no matter which choice we make.

Confluence is proved by examining expressions that can be rewritten in more than one way, and showing that no matter which way is chosen, further rewrites can get to the same point. Thus if $v_1 << v_2 << v_3$ are three variables, we can rewrite `(`v_3 `+ (`v_2 `+` v_1`))` in two different ways, but we can get to the same point, as shown by:

$$\begin{array}{lll} & \texttt{(}v_3\texttt{ + (}v_2\texttt{ + }v_1\texttt{))} & \\ \Longrightarrow & \texttt{(}v_3\texttt{ + (}v_1\texttt{ + }v_2\texttt{))} & \texttt{by commutativity 1} \\ \Longrightarrow & \texttt{(}v_1\texttt{ + (}v_3\texttt{ + }v_2\texttt{))} & \texttt{by commutativity 2} \\ \Longrightarrow & \texttt{(}v_1\texttt{ + (}v_2\texttt{ + }v_3\texttt{))} & \texttt{by commutativity 1} \\ & \texttt{(}v_3\texttt{ + (}v_2\texttt{ + }v_1\texttt{))} & \\ \Longrightarrow & \texttt{(}v_2\texttt{ + (}v_3\texttt{ + }v_1\texttt{))} & \texttt{by commutativity 2} \\ \Longrightarrow & \texttt{(}v_2\texttt{ + (}v_1\texttt{ + }v_3\texttt{))} & \texttt{by commutativity 1} \\ \Longrightarrow & \texttt{(}v_1\texttt{ + (}v_2\texttt{ + }v_3\texttt{))} & \texttt{by commutativity 2} \end{array}$$

Confluence is very important for both the speed and correctness of our algorithm. It is important for speed because we know we do not have to back up and try alternate paths to get a normal expression.

Third we must show that if $e_1 \Longrightarrow e_2$ then $e_1 = e_2$ is provable. That is, if e_1 rewrites to e_2 then we can prove that e_1 equals e_2. This is true of the algebra of simple

[4] Confluent algebras are also said to be '**Church Rosser**' because one of the first proofs of confluence was done by Church and Rosser. There are also two kinds of confluence, weak and strong, but for terminating rewrite rules they are the same.

expressions, and therefore we say that the rewrite rules of this algebra are '**sound**'. This is just a matter of proving that each rewrite rule becomes a provable equation if we replace $\Longrightarrow$ by $=$. We leave this to the reader.

Fourth and last, we must show that if e_1 and e_2 are both normal expressions, then $e_1 = e_2$ if and only if $e_1 \equiv e_2$. That is, normal expressions are provably equal if and only if they are syntactically identical. This is also true for the algebra of simple expressions, and therefore we say that the rewrite rules of this algebra are '**complete**'. We will prove this in Section 2.6.5.

Although we have discussed only the algebra of simple expressions, the following is valid for any formal algebra for which the rewrite rules are terminating, confluent, sound, and complete:

Equation Proof Algorithm. To prove an equation $e_1 = e_2$, rewrite each side of the equation to a normal expression, and compare the two normal expressions for syntactic identity. The original equation is provable if and only if the normal forms are syntactically identical.

Proof that the algorithm terminates, is efficient, and is correct: The algorithm terminates because the rewrite rules are terminating. It is efficient because the rules are confluent (choices do not matter and it is never necessary to back up and make another choice). It is correct because the rules are sound (the equations equal their normal forms) and complete (the normal forms are equal only if they are syntactically identical). QED

Of course there may be better or worse choices from the point of efficiency; all confluence proves is that one never has to back up. Algorithms that have to back up and try all choices typically have time complexity $O(2^n)$ for input of size n (say, with n operators), while algorithms that do not have to back up are more likely to have a time complexity like $O(n^2)$, and are much faster, even when they make the worst possible choices.

Exercise 2.1.1.1 Rewrite to normal form, naming the rewrite rules used at each step:

(a) $((s + u) + (t + w))$
(b) $((y + x) + z)$
(c) $((u + (s + s)) + r)$
(d) $(((z + y) + x) + w)$
(e) $(((z + y) + z) + y)$

Exercise 2.1.1.2 Use the equation proof algorithm to prove or disprove the following. Show all work: in particular, show all rewrites to normal form and name the rewrite rules used at each step.

(a) `((`x `+` y`) + (`z `+` w`)) = (`x `+ (`y `+ (`w `+` z`)))`
(b) `(`x `+ ((`w `+` z`) +` x`)) = ((`y `+` x`) + (`z `+` w`))`
(c) `(`x `+ (`y `+ (`z `+` y`))) = (((`z `+` x`) +` y`) +` y`)`
(d) `(`m `+ (`n `+ (`z `+` y`))) = (((`z `+` n`) +` y`) +` k`)`

2.1.2 A Normalization Function in LISP

To apply LISP to the problem of rewriting a simple expression to normal form we alter the syntax of expressions so they become LISP s-expressions with operators in prefix position that use symbols as variables. For the algebra of simple expressions, the grammar becomes:

simple-expression	::=	variable
	|	(+ simple-expression simple-expression)
variable	::=	symbol

The equational axioms become:

`(+ x y) = (+ y x)`	[**commutativity**]
`(+ (+ x y) z) = (+ x (+ y z))`	[**associativity**]

The rewrite rules become:

`(+` v_2 v_1`)` $\Longrightarrow$ `(+` v_1 v_2`)`	if $v_1 << v_2$	[**commutativity 1**]
`(+` v_2 `(+` v_1 e`))` $\Longrightarrow$ `(+` v_1 `(+` v_2 e`))`	if $v_1 << v_2$	[**commutativity 2**]
`(+ (+` e_1 e_2`)` e_3`)` $\Longrightarrow$ `(+` e_1 `(+` e_2 e_3`))`		[**associativity**]

where v_1, v_2 $\in$ variable; e, e_1, e_2, e_3 $\in$ simple-expression

where << denotes the dictionary ordering of symbols. An example applying these rewrite rules is:

	`(+ (+ s s) r)`	
$\Longrightarrow$	`(+ s (+ s r))`	by associativity
$\Longrightarrow$	`(+ s (+ r s))`	by commutativity 1
$\Longrightarrow$	`(+ r (+ s s))`	by commutativity 2

Code for a function named `normal-1` that implements these rules appears in Figures 2.1 and 2.2, and tests appear in Figure 2.3. The COMMONLISP `time` macro is used at the end of Figure 2.3 to determine the execution time of the evaluation of an expression that consists of a call to `normal-1`. The **run time** is the actual time taken by the computer, and may be significantly shorter than the **real time**, which is wall clock time and on a time shared computer may include the time of interruptions by other users of the computer.

This function works well, but in testing it, we noticed that it seemed to run a long time. We tried it on the following four expressions:

```
            (+ (+ (+ (+ u a) v) b) w)
        (+ (+ (+ (+ (+ (+ u a) v) b) w) c) x)
    (+ (+ (+ (+ (+ (+ (+ (+ u a) v) b) w) c) x) d) y)
(+ (+ (+ (+ (+ (+ (+ (+ (+ (+ u a) v) b) w) c) x) d) y) e) z)
```

and got the following results:

number of operators in expression	`normal-1` execution (run) time in seconds	ratio of successive times
4	0.00683	
		7.6
6	0.0517	
		6.2
8	0.319	
		5.8
10	1.78	

These suggest that as the number of operators in an expression of a particular type increases by 2, the execution time of `normal-1` increases by a <u>factor</u> of at least 5.8. Thus the execution time seems to be $O(5.8^{\frac{n}{2}})$ where n is the number of operators in the input expression.

This is a very poor result. The execution time is exponential. If one does a few examples by hand, one sees that probably the execution time should be $O(n^2)$, that is, proportional to the square of the length of the input.

```
;;;; Functions to put simple expressions in normal form.
;;;;
;;;; File:         normal-x.lsp
;;;; Author:       CS 51 (Bob Walton)
;;;; Version:      1

;; simple-expression ::= variable
;;                     | (+ simple-expression
;;                          simple-expression)
;; variable ::= symbol
;;
;;
;; v, v1, v2 are variables (symbols),
;; e, e1, e2, e3 are simple expressions
;;
;;
;; v1 << v2 iff v1 is lexically less than v2
;;
;;
;; ((e1 + e2) + e3) ===> (e1 + (e2 + e3))
;; (v2 + v1) ===> (v1 + v2) if v1 << v2
;; (v2 + (v1 + e3)) ===> (v1 + (v2 + e3)) if v1 << v2

;; Functions to make code more readable
;;
(defun op (e) (car e))      ; (op '(x y z)) ===> 'x
(defun arg1 (e) (cadr e))   ; (arg1 '(x y z)) ===> 'y
(defun arg2 (e) (caddr e))  ; (arg2 '(x y z)) ===> 'z
;;
(defun is+exp (e) (and (consp e) (eql '+ (op e))))
(defun isvar (e) (symbolp e))
(defun << (v1 v2) (string-lessp v1 v2))
```

Figure 2.1: `Normal-1`: Page 1 of 2

```
;; Function to find normal form of a simple expression.
;; First effort: just do it.
;;
(defun normal-1 (e)
  (cond
   ;; Base case:
   ((isvar e) e)           ; v ===> v

   ;; Be sure arguments are already normal:
   ;;
   ((or (not (equal (arg1 e) (normal-1 (arg1 e))))
        (not (equal (arg2 e) (normal-1 (arg2 e)))))
    (normal-1 `(,(op e) ,(normal-1 (arg1 e))
                        ,(normal-1 (arg2 e)))))

   ;; Associate to the right:
   ;;
   ((and (is+exp e)             ; (+ (+ e1 e2) e3) ===>
         (is+exp (arg1 e)))     ;   (+ e1 (+ e2 e3))
    (normal-1 `(+ ,(arg1 (arg1 e))
                  (+ ,(arg2 (arg1 e)) ,(arg2 e)))))

   ;; Sort variables:
   ;;
   ((and (is+exp e)             ; (+ v2 v1) ===>
         (isvar (arg1 e))       ;   (+ v1 v2)
         (isvar (arg2 e))       ;   if v1 << v2
         (<< (arg2 e) (arg1 e)))
    (normal-1 `(+ ,(arg2 e) ,(arg1 e))))

   ((and (is+exp e)             ; (+ v2 (+ v1 e3)) ===>
         (is+exp (arg2 e))      ;   (+ v1 (+ v2 e3))
         (isvar (arg1 e))       ;   if v1 << v2
         (isvar (arg1 (arg2 e)))
         (<< (arg1 (arg2 e)) (arg1 e)))
    (normal-1 `(+ ,(arg1 (arg2 e))
                  (+ ,(arg1 e) ,(arg2 (arg2 e))))))

   ;; Input expression is already normal.
   ;;
   (t e)))
```

Figure 2.2: `Normal-1`: Page 2 of 2

```
;;;; Test input for functions that put simple
;;;; expressions in normal form.
;;;;
;;;; File:         normal-x.in
;;;; Author:       CS 51 (Bob Walton)
;;;; Version:      3

(load "normal-x.lsp")
--->
;; Loading file normal-x.lsp ...
;; Loading of file normal-x.lsp is finished.
T

(normal-1 '(+ (+ x y) z))
--->
(+ X (+ Y Z))

(normal-1 '(+ (+ w x) (+ y z)))
--->
(+ W (+ X (+ Y Z)))

(normal-1 '(+ (+ (+ (+ (+ u v) w) x) y) z))
--->
(+ U (+ V (+ W (+ X (+ Y Z)))))

(normal-1 '(+ (+ (+ (+ (+ (+ u a) v) b) w) c) x))
--->
(+ A (+ B (+ C (+ U (+ V (+ W X))))))

(time (normal-1 '(+ (+ (+ (+ u a) v) b) w)))
--->

Real time: 0.006829f0 sec.
Run time: 0.006832f0 sec.
Space: 3552 Bytes
(+ A (+ B (+ U (+ V W))))
```

Figure 2.3: `Normal-1` Test

2.1.3 Instrumentation of the Normalization Function

We could try to find out what is going on by tracing the execution of `normal-1`, but often traces produce too much information that is hard to decipher. So we will instrument this function to get exactly the information we want.

We want several things. Our function searches for subexpressions in the input to which a rewrite rule can be applied, and when it finds such, applies a rule. We want to count the number of rules applied. We also want to count the number of times the function is called recursively on an already normal subexpression, in which case the function returns without applying any rule. Lastly, if a rule is applied, we want to be able to optionally print out the subexpression being rewritten and what it rewrote to.

Code that does all this is in Figures 2.4, 2.5, and 2.6, and a test of this code is in Figure 2.7.

We will not go into the details of the various COMMONLISP functions used in this code for the first time in this book, but will summarize them and how they work for us.

In Figure 2.4 the `defvar` expression is used to introduce three **global variables**. One, `*did-rewrite*`, counts the number of rewrites done. A second, `*did-nothing*`, counts the number of times our `normal` function was called with an already normal input, and could not find any rewrite to apply. The third, `*trace*`, is set to true to print out rewritten subexpressions and what they rewrite to whenever a rewrite occurs.

It is a venerable LISP convention that global variables are given names beginning and ending with '`*`'. This distinguishes them from **local variables**, which are variables introduced by '`defun`', '`let`', and '`do`'.

Returning to the code in Figure 2.4, the `rr` helper function defined there is to be called whenever a rewrite is actually done. This increments `*did-rewrite*` and, if `*trace*` is true, prints out the subexpression rewritten and the expression it rewrites to. In this function `setf` is used to set a global variable; `fresh-line` outputs an end of line character if there is anything on the current line, and does nothing if there is nothing on the current line; and `princ` prints an s-expression <u>without</u> preceding it with an end of line character.

The `no-rr` helper function is introduced in Figure 2.4 to record the failure to find any rewrite, and it simply increments `*did-nothing*`.

```
;; Instrumentation helper functions


;; Variables to gather statistics on calls to normal.
;;
(defvar *did-rewrite* 0) ; Number of rewrites done.
(defvar *did-nothing* 0) ; Number of times no rewrite
                         ; was found.
(defvar *trace* nil)     ; True to print trace,
                         ; false not to.

;; Functions to gather statistics and output trace.

;; Call when rewriting an original expression to a new
;; rewritten expression.
;;
(defun rr (original-expression rewritten-expression)
  (setf *did-rewrite* (+ *did-rewrite* 1))
  (cond (*trace*
         (fresh-line)
         (princ `(,original-expression =>
                  ,rewritten-expression))
         (fresh-line))))

;; Call when no applicable rewrite rule found.
;;
(defun no-rr ()
  (setf *did-nothing* (+ *did-nothing* 1)))
```

Figure 2.4: `Normal-2`: Page 1 of 3

```
;; Call (RR-TIME (NORMAL-... ...)) to test a
;; version of the normal rewriter.
;;
(defmacro rr-time (expression)
  `(let ((result nil))
    (setf *did-nothing* 0)
    (setf *did-rewrite* 0)
    (setf result (time ,expression))
    (fresh-line)
    (terpri)
    (princ '|Number of Rewrites: |)
    (princ *did-rewrite*)
    (terpri)
    (princ
     '|Number of failures to find any rewrite: |)
    (princ *did-nothing*)
    (terpri)
    (terpri)
    result))


;; Function to find normal form of a simple expression.
;;
;; Normal-1 plus instrumentation.
;;
(defun normal-2 (e)

  (cond
   ;; Base case:
   ;;
   ((isvar e)           ; v ===> v
    (no-rr)
    e)

   ;; Be sure arguments are already normal:
   ;;
   ((or (not (equal (arg1 e) (normal-2 (arg1 e))))
        (not (equal (arg2 e) (normal-2 (arg2 e)))))
    (normal-2 `(,(op e) ,(normal-2 (arg1 e))
                        ,(normal-2 (arg2 e)))))
```

Figure 2.5: Normal-2: Page 2 of 3

```
;; Associate to the right:
;;
((and (is+exp e)             ; (+ (+ e1 e2) e3) ===>
      (is+exp (arg1 e)))     ;   (+ e1 (+ e2 e3))
 (rr e `(+ ,(arg1 (arg1 e))
          (+ ,(arg2 (arg1 e)) ,(arg2 e))))
 (normal-2 `(+ ,(arg1 (arg1 e))
              (+ ,(arg2 (arg1 e)) ,(arg2 e)))))

;; Sort variables:
;;

((and (is+exp e)             ; (+ v2 v1) ===>
      (isvar (arg1 e))       ;   (+ v1 v2)
      (isvar (arg2 e))       ;   if v1 << v2
      (<< (arg2 e) (arg1 e)))
 (rr e `(+ ,(arg2 e) ,(arg1 e)))
 (normal-2 `(+ ,(arg2 e) ,(arg1 e))))

((and (is+exp e)             ; (+ v2 (+ v1 e3)) ===>
      (is+exp (arg2 e))      ;   (+ v1 (+ v2 e3))
      (isvar (arg1 e))       ;   if v1 << v2
      (isvar (arg1 (arg2 e)))
      (<< (arg1 (arg2 e)) (arg1 e)))
 (rr e `(+ ,(arg1 (arg2 e))
          (+ ,(arg1 e) ,(arg2 (arg2 e)))))
 (normal-2 `(+ ,(arg1 (arg2 e))
              (+ ,(arg1 e) ,(arg2 (arg2 e))))))

;; Input expression is already normal.
;;
(t
 (no-rr)
 e)))
```

Figure 2.6: `Normal-2`: Page 3 of 3

```
(setf *trace* t)
--->
T

(rr-time (normal-2 '(+ (+ (+ x a) y) b)))
--->
((+ X A) => (+ A X))
((+ X A) => (+ A X))
((+ (+ A X) Y) => (+ A (+ X Y)))
((+ X A) => (+ A X))
((+ X A) => (+ A X))
((+ (+ A X) Y) => (+ A (+ X Y)))
((+ (+ A (+ X Y)) B) => (+ A (+ (+ X Y) B)))
((+ (+ X Y) B) => (+ X (+ Y B)))
((+ Y B) => (+ B Y))
((+ Y B) => (+ B Y))
((+ X (+ B Y)) => (+ B (+ X Y)))
((+ (+ X Y) B) => (+ X (+ Y B)))
((+ Y B) => (+ B Y))
((+ Y B) => (+ B Y))
((+ X (+ B Y)) => (+ B (+ X Y)))

Real time: 0.01553f0 sec.
Run time: 0.01f0 sec.
Space: 20556 Bytes

Number of Rewrites: 15
Number of failures to find any rewrite: 106

(+ A (+ B (+ X Y)))

(setf *trace* nil)
--->
NIL
```

Figure 2.7: `Normal-2` Test

The `rr-time` macro is introduced in Figure 2.5 to function like an extension of the COMMONLISP `time` macro, which evaluates an expression and prints out the execution time of the evaluation. The extra functionality in `rr-time` initializes the counters `*did-rewrite*` and `*did-nothing*` to zero and prints them after the execution. In this macro we encounter for the first time the COMMONLISP `terpri` function that simply prints an end of line character.

In this macro we also encounter for the first time symbols with escaped characters, as in:

```
(princ '|Number of Rewrites: |)
```

Here '`|Number of Rewrites: |`' is a symbol that includes lower case letters, the '`:`' character, and blanks. These '**escaped characters**' are placed between a pair of '`|`'s. In general, a symbol name can have any number of '`|`' pairs, and any characters between such a pair are escaped characters. Escaped lower case letters are <u>not</u> converted to upper case, and any printing character except '`|`' can be included in the symbol as an escaped character. Spaces can also be escaped characters.

For example:

`|Number of Rewrites: |` and `n|umber of |r|ewrites: |`

are two different ways of inputting the same symbol: in the second input, the unescaped letters '`n`' and '`r`' are converted to upper case. To take another example, '`||`' and '`||||`' are both ways of inputing the same symbol that has no characters at all in its name.

The rewrite of `normal-1` that uses these instrumentation helper functions is called `normal-2`, and appears in Figures 2.5 and 2.6. Note that `rr` and `no-rr` are executed because they do something other than produce a value, and these expressions appear just before expressions that compute a value to be returned by some `cond`.

In general, wherever LISP expects an expression that is to be computed to produce a value that is to be returned from a special function like '`cond`', '`defun`', or '`let`', you can write a list of expressions instead of a single expression. All the expressions will be evaluated and the value of the last will be the value returned. The other expressions need to do something besides simply produce a value, or they would be useless. The things an expression does besides produce a value are called the '**side effects**' of the expression, so all the expressions but the last in the list should have side effects. In our case, `rr` and `no-rr` are evaluated for their side effects, not for their value.

A test of `normal-2` is given in Figure 2.7.

This test reveals that rewrites are being repeated. For example, the first three rewrites are repeated as a group, and the last four rewrites are repeated as a group.

The culprit is most likely the code:

```
;; Be sure arguments are already normal:
;;
((or (not (equal (arg1 e) (normal-2 (arg1 e))))
     (not (equal (arg2 e) (normal-2 (arg2 e)))))
(normal-2 `(,(op e) ,(normal-2 (arg1 e))
                    ,(normal-2 (arg2 e)))))
```

In this code `(normal-2 (arg1 e))` is evaluated twice if any rewrites are performed during it, and the same may be true of `(normal-2 (arg2 e))`. The effect of this is roughly that the execution time on an input is twice the execution time on the subexpressions of the input plus a constant, and this gives a time exponential in the size of the input.[5]

Exercise 2.1.3.1 Explain as best you can why increasing the number of operators by 2 in the examples just given increases the execution time by a factor of 5.8 and not just by a factor of 4.

2.1.4 Optimization of the Normalization Function

One fix for the problem just uncovered is to divide `normal-2` up into two functions, a first '`normal`' function that does everything up to the point were it is discovered that the input has an operator and two operands, and a second '`apply-rules`' function that works on expressions with an operator and two already normal operands. The first '`normal`' function, when it finds it has an operator and two operands in its input, calls itself recursively to make the operands normal, and then calls the second '`apply-rules`' function with an expression consisting of the operator and the two

[5] To go into somewhat more detail, in going from an expression e to an expression (`+` e v) we normalize e twice and add one `+` operator and one variable v to the expression. Thus we appear to almost double the execution time while increasing the number of operators by 1. So we would expect that increasing the number of operators by 2 in this way would multiply the execution time by 4. This rough estimate is a bit below the 5.8 we found experimentally.

normal operands.

Code written in this way appears in Figures 2.8 and 2.9. The `apply-rules` function in this code takes an expression consisting of an operator and two normal operands, searches for rules to apply to this expression, and applies the rules. Because a subexpression of a normal expression is itself normal, when the rules are applied, the `apply-rules` function can call itself recursively to ensure that the final results are normal, rather than having to call the full `normal` function. If we did not do this, we would be repeatedly checking the same normal subexpressions to see if they are normal, and would probably still have an exponential time computation, even though it would run faster than `normal-2`.

Some results for `normal-3` appear in Figure 2.10. These show an absence of the rewrite rule repetitions that characterized the results for `normal-2` in Figure 2.7.

Timing tests on the same four expressions tested for `normal-1` at the end of Section 2.1.2 give the following results for `normal-2` and `normal-3`:

	normal-2				normal-3			
N	T_2	$\frac{T_2}{5.8^{\frac{N}{2}}}$	RR_2	F_2	T_3	$\frac{T_3}{N^2}$	RR_3	F_3
4	0.00878	0.000261	37	298	0.000976	0.000061	9	16
6	0.0703	0.000360	265	2298	0.00195	0.000054	21	31
8	0.391	0.000346	1553	14002	0.00390	0.000061	38	51
10	2.15	0.000328	8225	75898	0.00586	0.000059	60	76

N is the number of operators in the input expression (argument)
T_j is the running time in seconds for `normal-`j
RR_j is the number of rewrites for `normal-`j
F_j is the number of failures to find any rewrite for `normal-`j

It can be seen that while the execution time of `normal-2` is proportional to $5.8^{\frac{N}{2}}$, the execution time of `normal-3` is proportional to N^2, for the type of expression tested. Some experiments in doing rewrites by hand suggest that an execution time proportional to N^2 is likely to be the best one can do.

Exercise 2.1.4.1 Explain why the execution time of `normal-3` can be expected to be proportional to N^2, where N is the number of operators in the input.

```
;; Function to find normal form of a simple expression.
;;
;; This version is efficient because it does not
;; unnecessarily check subexpressions to see if they
;; are normal.  It uses a helper function, apply-rules,
;; that takes as input an expression of the form
;; (+ e1 e2) where e1 and e2 are already known to be
;; normal.
;;
(defun normal-3 (e)

  (cond
   ;; Base case:
   ;;
   ((isvar e)           ; v ===> v
    (no-rr)
    e)

   ;; Apply rules to operator with normal arguments.
   ;;
   (t
    (apply-rules-3
     `(,(op e)
        ,(normal-3 (arg1 e))
        ,(normal-3 (arg2 e)))))))
```

Figure 2.8: `Normal-3`: Page 1 of 2

```
;; Apply rules assuming expression has form (+ e1 e2)
;; where e1 and e2 are normal.
;;
(defun apply-rules-3 (e)
  (cond

   ;; Associate to the right:
   ;;
   ((and (is+exp e)             ; (+ (+ e1 e2) e3) ===>
         (is+exp (arg1 e)))     ;   (+ e1 (+ e2 e3))
    (rr e `(+ ,(arg1 (arg1 e))
              (+ ,(arg2 (arg1 e)) ,(arg2 e))))
    (apply-rules-3
     `(+ ,(arg1 (arg1 e))
         ,(apply-rules-3 `(+ ,(arg2 (arg1 e)) ,(arg2 e))))))

   ;; Sort variables:
   ;;
   ((and (is+exp e)             ; (+ v2 v1) ===>
         (isvar (arg1 e))       ;   (+ v1 v2)
         (isvar (arg2 e))       ;   if v1 << v2
         (<< (arg2 e) (arg1 e)))
    (rr e `(+ ,(arg2 e) ,(arg1 e)))
    (apply-rules-3 `(+ ,(arg2 e) ,(arg1 e))))

   ((and (is+exp e)             ; (+ v2 (+ v1 e3)) ===>
         (is+exp (arg2 e))      ;   (+ v1 (+ v2 e3))
         (isvar (arg1 e))       ;   if v1 << v2
         (isvar (arg1 (arg2 e)))
         (<< (arg1 (arg2 e)) (arg1 e)))
    (rr e `(+ ,(arg1 (arg2 e))
              (+ ,(arg1 e) ,(arg2 (arg2 e)))))
    (apply-rules-3
     `(+ ,(arg1 (arg2 e))
         ,(apply-rules-3 `(+ ,(arg1 e) ,(arg2 (arg2 e)))))))

   ;; Input expression is already normal.
   ;;
   (t (no-rr) e)))
```

Figure 2.9: Normal-3: Page 2 of 2

```
(setf *trace* t)
--->
T

(rr-time (normal-3 '(+ (+ (+ x a) y) b)))
--->
((+ X A) => (+ A X))
((+ (+ A X) Y) => (+ A (+ X Y)))
((+ (+ A (+ X Y)) B) => (+ A (+ (+ X Y) B)))
((+ (+ X Y) B) => (+ X (+ Y B)))
((+ Y B) => (+ B Y))
((+ X (+ B Y)) => (+ B (+ X Y)))

Real time: 0.004161f0 sec.
Run time: 0.0f0 sec.
Space: 9288 Bytes

Number of Rewrites: 6
Number of failures to find any rewrite: 11

(+ A (+ B (+ X Y)))

(setf *trace* nil)
--->
NIL
) )
```

Figure 2.10: `Normal-3` Test

2.2 Algebras of Integer Evaluation

We next consider the formal algebra whose expressions are made from integers and the operators `+` and `*` (times). The grammar is:

$$
\begin{array}{lcl}
\text{integer-expression} & ::= & \text{integer} \\
 & | & (\ \text{integer-expression} \ \texttt{+} \ \text{integer-expression}\ ) \\
 & | & (\ \text{integer-expression} \ \texttt{*} \ \text{integer-expression}\ )
\end{array}
$$

Here integers can be positive or negative, so that '`( -4 + -5 )`' is an expression of the algebra.

The equational axioms are:

$$
\begin{array}{ll}
\texttt{(}i_1 \ \texttt{+} \ i_2\texttt{)} \ \texttt{=} \ i_3 & \text{if } i_3 = i_1 + i_2 \text{ mathematically} \\
\texttt{(}i_1 \ \texttt{*} \ i_2\texttt{)} \ \texttt{=} \ i_3 & \text{if } i_3 = i_1 \cdot i_2 \text{ mathematically}
\end{array}
$$

where i_1, i_2, $i_3 \in$ integer

These evaluate the application of an operator to two integers. They are actually meta-axioms with i_1, i_2, and i_3 being meta-variables and each of the above meta-axioms standing for an infinite number of axioms such as:

```
(0 + 0) = 0
(0 + 1) = 1
...
(1 + 0) = 1
(1 + 1) = 2
(1 + 2) = 3
...
```

Thus in '$\texttt{(}i_1 \ \texttt{+} \ i_2\texttt{)}$' the `+` is a symbol in an expression of the algebra, whereas in '$i_3 = i_1 + i_2$' the $+$ is a mathematical operation between integers interpreted as in mathematics.

The rewrite rules are just directed versions of the axioms:

$$\begin{array}{lll} (i_1 \texttt{ + } i_2) & \Longrightarrow i_3 & \text{if } i_3 = i_1 + i_2 \text{ mathematically} \\ (i_1 \texttt{ * } i_2) & \Longrightarrow i_3 & \text{if } i_3 = i_1 \cdot i_2 \text{ mathematically} \end{array}$$

where $i_1,\ i_2,\ i_3 \in$ integer

It can be shown that these rewrite rules are terminating (each rewrite reduces the number of operators in an expression), confluent, sound (the rewrite rules are just directed versions of the axioms), and complete (with the normal forms being just the integers). A LISP program written to perform these rewrites is called an **evaluator** because it simply evaluates subexpressions in which all the operands are integers.

We will call the formal algebra we have just introduced the **algebra of integer evaluation**.

Now suppose we add to this algebra the equational axioms:

$$i_1 \texttt{ = } i_2 \quad \text{if } i_1 = i_2 + 4 \text{ mathematically}$$

where $i_1,\ i_2 \in$ integer

and the corresponding rewrite rules:

$$\begin{array}{lll} i_1 & \Longrightarrow i_2 & \text{if } i_2 = i_1 - 4 \text{ and } i_1 \geq 4 \text{ mathematically} \\ i_1 & \Longrightarrow i_2 & \text{if } i_2 = i_1 + 4 \text{ and } i_1 < 0 \text{ mathematically} \end{array}$$

where $i_1,\ i_2 \in$ integer

The normal forms are now the integers `0`, `1`, `2`, and `3`. This algebra is the **algebra of evaluation modulo 4**. It can be shown that its rewrite rules are terminating, confluent, sound, and complete. In this algebra:

$$e_1 \texttt{ = } e_2 \qquad \text{if and only if} \qquad e_1 = e_2 \text{ (modulo 4) mathematically}$$

That is, a formal equation of the algebra is true if and only if the two sides evaluate mathematically to the same integer modulo 4. Two integers are the same modulo 4 if and only if their difference is an exact multiple of 4.

To prove that the algebra of evaluation modulo 4 is confluent one must handle situations such as:

```
(7 + 2) ⟹ 9 ⟹ 5 ⟹ 1
(7 + 2) ⟹ (3 + 2) ⟹ 5 ⟹ 1
```

In general, if we write 'i mod 4' for the integer 0, 1, 2, or 3 that equals i modulo 4, then the following facts are useful for proving confluence:

$$\texttt{((}i_1\texttt{ mod 4) + (}i_2\texttt{ mod 4)) mod 4 = (}i_1\texttt{ + }i_2\texttt{) mod 4}$$
$$\texttt{((}i_1\texttt{ mod 4) * (}i_2\texttt{ mod 4)) mod 4 = (}i_1\texttt{ * }i_2\texttt{) mod 4}$$

The algebra of evaluation modulo 4 can be shown to be terminating, confluent, sound, and complete. Also, we could have used any integer greater than 0 in place of 4.

Exercise 2.2.1 In the algebra of evaluation modulo 4, give all possible rewrites of the following (as was done for '`(7 + 2)`' above). You need not name the rewrite rules used. To save space, you may if you like give the result as a graph with $\Longrightarrow$ arrows and nodes labeled by expressions.

(a) `(9 + 5)`
(b) `((3 + 2) + 4)`
(c) `(6 * 2)`
(d) `((3 * 2) + 4)`

2.3 The Algebra of Polynomials

If we permit `+`, `*`, numbers, and variables in our expression we get the **algebra of polynomials**. The grammar is:

$$\begin{array}{lcl} \text{expression} & ::= & \text{number} \mid \text{variable} \\ & \mid & (\text{ expression } \texttt{+} \text{ expression }) \\ & \mid & (\text{ expression } \texttt{*} \text{ expression }) \\ \text{variable} & ::= & x \mid y \mid z \mid x_1 \mid y_1 \mid z_1 \mid \ldots \end{array}$$

where a number can be any real number (integer, ratio, or floating point). The equational axioms are:

`(`x` + `y`) = (`y` + `x`)`	[**commutativity**]
`(`x` * `y`) = (`y` * `x`)`	[**commutativity**]
`((`x` + `y`) + `z`) = (`x` + (`y` + `z`))`	[**associativity**]
`((`x` * `y`) * `z`) = (`x` * (`y` * `z`))`	[**associativity**]
`(`n_1` + `n_2`) = `n_3	if $n_3 = n_1 + n_2$ mathematically
`(`n_1` * `n_2`) = `n_3	if $n_3 = n_1 \cdot n_2$ mathematically
`(0 + `x`) = `x	[**zero**]
`(1 * `x`) = `x	[**unity**]
`(0 * `x`) = 0`	[**nullity**]
`((`x` + `y`) * `z`) = ((`x` * `z`) + (`y` * `z`))`	[**distributivity**]

where n_1, n_2, $n_3 \in$ number

An example use of these axioms is:

	`(x + ((5 + x) + 10))`	
=	`(x + (5 + (x + 10)))`	by associativity
=	`(x + (5 + (10 + x)))`	by commutativity
=	`(x + ((5 + 10) + x))`	by associativity
=	`(x + (15 + x))`	by combining numbers
=	`((x + 15) + x)`	by associativity
=	`((15 + x) + x)`	by commutativity
=	`(15 + (x + x))`	by associativity
=	`(15 + ((1 * x) + x))`	by unity
=	`(15 + ((1 * x) + (1 * x)))`	by unity
=	`(15 + ((1 + 1) * x))`	by distributivity
=	`(15 + (2 * x))`	by combining numbers

We will now introduce rewrite rules that will put expressions into a normal form. Examples of rewrites we want to occur are:

$$-4 + 4c + 4 + 9b + -4c + d + -8b \quad \Longrightarrow \quad b + d$$

$$(a+b)(c+d) \quad \Longrightarrow \quad a(c+d) + b(c+d) \quad \Longrightarrow \quad ac + ad + bc + bd$$

where we have abbreviated expressions by omitting parentheses and multiplication signs. We want our normal forms to be polynomials. A polynomial (e.g., $5 + x + xy$)

is a sum of monomials; a monomial (e.g. 5 or x or xy) is a product of factors; and a factor (e.g. 5 or x or y) is a number or a variable. We want the factors in a normal monomial to be sorted (e.g., xy and not yx) and we want there to be at most one numeric factor (the numeric factors must be combined). We want the monomials in a normal polynomial to be sorted (e.g. $xy + yy$ and not $yy + xy$) and we want there to be at most one monomial with each given set of variables (monomials with equal variable sets are combined: e.g., $5xy + 3xy \Longrightarrow 8xy$). With these ideas in mind, we begin the search for rewrite rules.

We want our normal expressions to be right associative, and therefore introduce the rules:

$$\begin{array}{lll} \texttt{((}e_1 \texttt{ + } e_2\texttt{) + } e_3\texttt{)} \Longrightarrow \texttt{(}e_1 \texttt{ + (}e_2 \texttt{ + } e_3\texttt{))} & & [\textbf{associativity}] \\ \texttt{((}e_1 \texttt{ * } e_2\texttt{) * } e_3\texttt{)} \Longrightarrow \texttt{(}e_1 \texttt{ * (}e_2 \texttt{ * } e_3\texttt{))} & & [\textbf{associativity}] \end{array}$$

where e_1, e_2, $e_3 \in$ expression

We also want all * operators to be inside all + operators, which we can do using distributivity:

$$\begin{array}{lll} \texttt{((}e_1 \texttt{ + } e_2\texttt{) * } e_3\texttt{)} \Longrightarrow \texttt{((}e_1 \texttt{ * } e_3\texttt{) + (}e_2 \texttt{ * } e_3\texttt{))} & & [\textbf{distributivity}] \\ \texttt{(}e_1 \texttt{ * (}e_2 \texttt{ + } e_3\texttt{))} \Longrightarrow \texttt{((}e_1 \texttt{ * } e_2\texttt{) + (}e_1 \texttt{ * } e_3\texttt{))} & & [\textbf{distributivity}] \end{array}$$

where e_1, e_2, $e_3 \in$ expression

An example of using the rewrite rules we have introduced so far is:

$$\begin{array}{lll} & \texttt{(((}s \texttt{ + } r\texttt{) * } t\texttt{) * } v\texttt{)} & \\ \Longrightarrow & \texttt{((}s \texttt{ + } r\texttt{) * (}t \texttt{ * } v\texttt{))} & \text{by associativity} \\ \Longrightarrow & \texttt{((}s \texttt{ * (}t \texttt{ * } v\texttt{)) + (}r \texttt{ * (}t \texttt{ * } v\texttt{)))} & \text{by distributivity} \end{array}$$

If we introduce the grammar categories:

$$\begin{array}{lll} \text{factor} & ::= & \text{number} \mid \text{variable} \\ \text{monomial} & ::= & \text{factor} \\ & \mid & \texttt{(} \text{ factor } \texttt{*} \text{ monomial } \texttt{)} \\ \text{polynomial} & ::= & \text{monomial} \\ & \mid & \texttt{(} \text{ monomial } \texttt{+} \text{ polynomial } \texttt{)} \end{array}$$

then the rewrite rules we have introduced so far permit us to rewrite any expression into a polynomial. Here a polynomial is a right associative sum of monomials, and a monomial is a right associative product of factors.

The next step is to sort the factors in a monomial. We order factors so that two variables are in lexical order and all numbers are before all variables. Thus $x << y$ and $5 << a$. With this order of factors, we give the following rewrite rules for sorting factors:

$$\begin{array}{lll} (f_2 \;\texttt{*}\; f_1) \Longrightarrow (f_1 \;\texttt{*}\; f_2) & \text{if } f_1 << f_2 & [\textbf{commutativity}] \\ (f_2 \;\texttt{*}\; (f_1 \;\texttt{*}\; e)) \Longrightarrow (f_1 \;\texttt{*}\; (f_2 \;\texttt{*}\; e)) & \text{if } f_1 << f_2 & [\textbf{commutativity}] \end{array}$$

where $f_1,\ f_2 \in$ factor; $e \in$ expression

Rather than try to sort numbers, we **combine numbers**, using the rewrite rules:

$$\begin{array}{ll} (n_1 \;\texttt{*}\; n_2) \Longrightarrow n_3 & \text{if } n_3 = n_1 \cdot n_2 \text{ mathematically} \\ (n_1 \;\texttt{*}\; (n_2 \;\texttt{*}\; e)) \Longrightarrow (n_3 \;\texttt{*}\; e) & \text{if } n_3 = n_1 \cdot n_2 \text{ mathematically} \end{array}$$

where $n_1,\ n_2,\ n_3 \in$ number; $e \in$ expression

We also have special rules to simplify multiplication by 0 and 1:

$$\begin{array}{ll} (\texttt{1} \;\texttt{*}\; e) \Longrightarrow e & [\textbf{unity}] \\ (\texttt{0} \;\texttt{*}\; e) \Longrightarrow \texttt{0} & [\textbf{nullity}] \end{array}$$

where $e \in$ expression

An example applying the rewrite rules for sorting factors and combining numbers is:

$$\begin{array}{lll} & (s \;\texttt{*}\; (\texttt{5} \;\texttt{*}\; (t \;\texttt{*}\; \frac{1}{5}))) & \\ \Longrightarrow & (\texttt{5} \;\texttt{*}\; (s \;\texttt{*}\; (t \;\texttt{*}\; \frac{1}{5}))) & \text{by commutativity} \\ \Longrightarrow & (\texttt{5} \;\texttt{*}\; (s \;\texttt{*}\; (\frac{1}{5} \;\texttt{*}\; t))) & \text{by commutativity} \\ \Longrightarrow & (\texttt{5} \;\texttt{*}\; (\frac{1}{5} \;\texttt{*}\; (s \;\texttt{*}\; t))) & \text{by commutativity} \\ \Longrightarrow & (\texttt{1} \;\texttt{*}\; (s \;\texttt{*}\; t)) & \text{by combining numbers} \\ \Longrightarrow & (s \;\texttt{*}\; t) & \text{by unity} \end{array}$$

Lastly we want to sort the monomials in a polynomial. To do this we must order monomials. For monomials m_1 and m_2 we define $m_1 << m_2$ to mean that either the number of variable occurrences in m_1 is less than the number of variable occurrences in m_2, or the numbers are equal and the list of variables in m_1 precedes the list of variables in m_2 in dictionary order. Thus:

x << y	because	$x << y$
`5` << a	because	$0 < 1$ variable occurrences
x << `(`a `*` b`)`	because	$1 < 2$ variable occurrences
`(5 *` x`)` << y	because	$x << y$
`(`x `*` y`)` << `(`y `*` y`)`	because	$x\,y << y\,y$
`(5 * (`x `*` y`))` << `(`y `*` y`)`	because	$x\,y << y\,y$

Note that numeric factors are ignored in ordering monomials.

Given this ordering of monomials we introduce rewrite rules for sorting monomials in a polynomial:

$$(m_2 \texttt{ + } m_1) \Longrightarrow (m_1 \texttt{ + } m_2) \quad \text{if } m_1 << m_2 \quad [\textbf{commutativity}]$$
$$(m_2 \texttt{ + } (m_1 \texttt{ + } e)) \Longrightarrow (m_1 \texttt{ + } (m_2 \texttt{ * } e)) \quad \text{if } m_1 << m_2 \quad [\textbf{commutativity}]$$

where m_1, $m_2 \in$ monomial; $e \in$ expression

As with factors, we **combine numbers** rather than try to sort them.

$$((n_1 \texttt{ * } m) \texttt{ + } (n_2 \texttt{ * } m)) \Longrightarrow (n_3 \texttt{ * } m) \quad \text{if } n_3 = n_1 + n_2 \text{ mathematically}$$
$$((n_1 \texttt{ * } m) \texttt{ + } ((n_2 \texttt{ * } m) \texttt{ + } e)) \Longrightarrow ((n_3 \texttt{ * } m) \texttt{ + } e)$$
$$\text{if } n_3 = n_1 + n_2 \text{ mathematically}$$

where n_1, n_2, $n_3 \in$ number; $m \in$ monomial; $e \in$ expression

We have to pay special attention to monomials like '`(`x `*` y`)`' that have an implicit numeric factor of 1, and to monomials like '`5`' that have no variables. This gives us many special cases of the previous two rewrite rules:

$$
\begin{array}{ll}
\texttt{(}n_1 \texttt{ + } n_2\texttt{)} \Longrightarrow n_3 & \text{if } n_3 = n_1 + n_2 \text{ mathematically} \\
\texttt{(}m \texttt{ + (}n_2 \texttt{ * } m\texttt{))} \Longrightarrow \texttt{(}n_3 \texttt{ * } m\texttt{)} & \text{if } n_3 = 1 + n_2 \text{ mathematically} \\
\texttt{((}n_1 \texttt{ * } m\texttt{) + } m\texttt{)} \Longrightarrow \texttt{(}n_3 \texttt{ * } m\texttt{)} & \text{if } n_3 = n_1 + 1 \text{ mathematically} \\
\texttt{(}m \texttt{ + } m\texttt{)} \Longrightarrow \texttt{(2 * } m\texttt{)} & \\
\texttt{(}n_1 \texttt{ + (}n_2 \texttt{ + } e\texttt{))} \Longrightarrow \texttt{(}n_3 \texttt{ + } e\texttt{)} & \text{if } n_3 = n_1 + n_2 \text{ mathematically} \\
\texttt{(}m \texttt{ + ((}n_2 \texttt{ * } m\texttt{) + } e\texttt{))} \Longrightarrow \texttt{((}n_3 \texttt{ * } m\texttt{) + } e\texttt{)} & \text{if } n_3 = 1 + n_2 \text{ mathematically} \\
\texttt{((}n_1 \texttt{ * } m\texttt{) + (}m \texttt{ + } e\texttt{))} \Longrightarrow \texttt{((}n_3 \texttt{ * } m\texttt{) + } e\texttt{)} & \text{if } n_3 = n_1 + 1 \text{ mathematically} \\
\texttt{(}m \texttt{ + (}m \texttt{ + } e\texttt{))} \Longrightarrow \texttt{((2 * } m\texttt{) + } e\texttt{)} &
\end{array}
$$

where n_1, n_2, $n_3 \in$ number; $m \in$ monomial; $e \in$ expression

Lastly we have the rule for removing the 0 monomial:

$$\texttt{(0 + } e\texttt{)} \Longrightarrow e \qquad [\textbf{zero}]$$

where $e \in$ expression

An example applying the rewrite rules for sorting and combining monomials is:

$$
\begin{array}{lll}
 & \texttt{((}x \texttt{ * } y\texttt{) + (}x \texttt{ + (5 * (}x \texttt{ * } y\texttt{))))} & \\
\Longrightarrow & \texttt{(}x \texttt{ + ((}x \texttt{ * } y\texttt{) + (5 * (}x \texttt{ * } y\texttt{))))} & \text{by commutativity} \\
\Longrightarrow & \texttt{(}x \texttt{ + (6 * (}x \texttt{ * } y\texttt{)))} & \text{by combining numbers}
\end{array}
$$

These are all the rewrite rules we need for the algebra of polynomials. The rewrite rules for this algebra are terminating, sound, confluent, and complete. To be confluent and complete we must require that all mathematical computation with numbers be exact, which means that numbers must be integers or ratios if they are to be represented finitely in a typical manner.

If we do not insist that all numbers be integers or ratios, and decide to let some numbers in our expressions be finite precision floating point, we must amend some of the axioms and rewrite rules so that the results of 'mathematical' operations are rounded. Thus for a precision of around 16 decimal places, we get the rewrite rule:

$$\texttt{1e-20 + 1.0} \Longrightarrow \texttt{1.0}$$

From this we derive that

$$
\begin{aligned}
\texttt{((1e-20 + 1.0) - 1.0)} &\Longrightarrow \texttt{(1.0 + -1.0)}\\
&\Longrightarrow \texttt{0.0}\\
\texttt{((1e-20 + 1.0) - 1.0)} &\Longrightarrow \texttt{(1e-20 + (1.0 - 1.0))}\\
&\Longrightarrow \texttt{(1e-20 + 0.0)}\\
&\Longrightarrow \texttt{1e-20}
\end{aligned}
$$

Thus there are two legal rewrites of `((1e-20 + 1.0) - 1.0)` that lead to distinct normal forms, `0.0` and `1e-20` (numbers are normal forms), and so the rewrite system is not confluent. This is a standard problem with floating point arithmetic, namely that the order in which arithmetic operations are performed can have noticeable impact on the answers.

In our case the impact is noticeable if you are testing for zero. You might think that `1e-20` is very small, and therefore you can program the computer to treat it as zero, but to do this you require special knowledge about what is small. Since floating point calculations in which all meaningful numbers are near `1e-20` are useful, you cannot just declare `1e-20` to be zero all the time.

Exercise 2.3.1 Prove the following, naming the axioms used at each step:

(a) $((x + y) * (z + w)) = (((w + z) * x) + ((w * y) + (z * y)))$
(b) $((x + x) + (y * y))) = ((y * y) + (2 * x))$
(c) $((x * (y + z)) + 5)) = ((5 + (z * x)) + (x * y))$

Exercise 2.3.2 Rewrite to normal form in the algebra of polynomials, naming the rewrite rules used at each step:

(a) $(((x + y) * (x + y)) + ((3 * y) * x))$
(b) $(x + (y + (x + (b + y))))$
(c) $((x * (x + y)) + ((x + x) * y))$

Exercise 2.3.3 In the algebra of polynomials, give all possible rewrites of the following to normal form. You need not name the rewrite rules used. To save space, you may if you like give the result as a graph with $\Longrightarrow$ arrows and nodes labeled by expressions.

(a) $(((x * y) + x) + (5 * (x * y)))$
(b) $((d * (5 * c)) * b)$
(c) $(x + (y + ((5 * x) + (3 * x))))$

Exercise 2.3.4 Use the equation proof algorithm to prove or disprove the following. Show all work: in particular, show all rewrites to normal form and name the rewrite rules used at each step.

(a) `(`x` + (`x` + (2 * `y`))) = (`y` + (`y` + (2 * `x`)))`
(b) `(((`y` * 3) * `x`) + (`y` + (`x` * `y`))) =`
`(((`x` * (`y` * 2)) + ((3 * `x`) * y)) + `y`)`
(c) `((`x` + `x`) + (`y` * `y`))) = (`x` + (`x` + (2 * `y`)))`
(d) `((`x` + `y`) * (`x` + `z`)) = ((`z` + `x`) * (`x` + `y`))`

2.4 The Algebra of Rational Expressions

In this section we investigate what happens when we add negation and division to the algebra of polynomials. Division makes things rather messy. The purpose of this section is to exemplify what happens when the messiness of the real world impinges on a perfectly elegant theory. The reader can skip or skim this section.

With the addition of negation and division we get the **algebra of rational expressions**. The grammar is:

```
expression ::= number | variable
             | ( expression + expression )
             | ( expression - expression )
             | ( expression * expression )
             | ( expression / expression )
```

The equational axioms are the same as those of the algebra of polynomials with the addition of:

`(x - y) = (x + (-1 * y))`		[**negation**]
`(x + (y / z)) = (((x * z) + y) / z)`		[**+/ distributivity**]
`((x / y) + z) = ((x + (y * z)) / y)`		[**+/ distributivity**]
`((x / y) * z) = ((x * z) / y)`		[***/ distributivity**]
`(x * (y / z)) = ((x * y) / z)`		[***/ distributivity**]
`((x / y) / z) = (x / (y * z))`		[**/ combination**]
`(x / (y / z)) = ((x * z) / y)`	if not z `= 0`	[**/ combination**]
`(x / (y / 0)) = (0 / 0)`		[**/ combination**]
`(x / x) = 1`	if not x `= 0`	[**common factors**]
`(x / 0) = (0 / 0)`		[**divide by 0**]
`(x / 1) = x`		[**divide by 1**]

There are some difficulties here with division by zero. To solve these we introduce the notion that dividing by zero produces a special value, '**undefined**'. Any operation with an operand that equals the undefined value produces an undefined value as its result. In the formal algebra, `(0 / 0)` can be used to represent the undefined value.

We must be careful not to allow axioms from which we could prove `(1 / (1 / 0)) = ((1 * 0) / 1)`, since the left side is undefined but the right side is 0. Thus we have to require that $z \neq 0$ in one of the `/` combination axioms.

Next we want to introduce rewrite rules that will rewrite rational expressions into normal forms. Some examples of the rewrites we would like are:

$$\frac{a}{b} + \frac{c}{d} \implies \frac{\frac{a}{b} \cdot d + c}{d} \implies \frac{\frac{ad+bc}{b}}{d} \implies \frac{ad+bc}{bd}$$

$$1 + \frac{1}{1 + \frac{1}{1+\frac{1}{1+x}}} \implies \frac{5+3x}{3+2x}$$

$$\frac{(x+1)(x+3)}{(x+2)(x+4)} - \frac{(y+1)(y+3)}{(y+2)(y+4)} \implies \frac{14x - 14y + 5x^2 - 5y^2 + 2x^2y - 2xy^2}{64 + 48x + 8x^2 + 36xy + 8y^2 + 6x^2y + 6xy^2 + x^2y^2}$$

where we have abbreviated expressions by omitting parentheses and multiplication signs. We want our normal forms to be quotients of two polynomials, or a single polynomial by itself. The polynomials should be normal forms in the algebra of polynomials discussed above.

However, if we are not careful, our rewrite rules will not be terminating, as is illustrated by the following example:

$$\frac{x^n}{x+y} \cdot (x+y) \implies \frac{x^{n+1}}{x+y} + \frac{x^n \cdot y}{x+y} \implies \frac{\frac{x^{n+1}}{x+y} \cdot (x+y) + x^n \cdot y}{x+y} \implies \ldots$$

To overcome this difficulty, we restrict the `*+` distributivity rewrite rules so they can only apply within a polynomial that contains no divisions:

`((`p_1 `+` p_2`)` `*` p_3`)`	$\implies$	`((`p_1 `*` p_3`)` `+` `(`p_2 `*` p_3`))`	[**distributivity**]
`(`p_1 `*` `(`p_2 `+` p_3`))`	$\implies$	`((`p_1 `*` p_2`)` `+` `(`p_1 `*` p_3`))`	[**distributivity**]

where p_1, p_2, $p_3 \in$ polynomial

In the algebra of rational expressions we adopt all the rewrite rules of the algebra of polynomials verbatim, except for the distributivity rules, for which we adopt the modified versions just given. But of course the algebra of rational expressions will have additional rules that we now introduce.

Just as the normal distributivity law moves `*` inside `+`, we want to use the `+/` and `*/` distributivity laws to move `+` and `*` inside `/`. But if we are not careful, our rewrite rules will not be confluent, as is illustrated by the following example:

$$w + \frac{x}{\frac{y}{z}} \Longrightarrow \frac{w \cdot \frac{y}{z} + x}{\frac{y}{z}} \Longrightarrow \frac{\frac{w \cdot y}{z} + x}{\frac{y}{z}} \Longrightarrow \frac{\frac{w \cdot y + x \cdot z}{z}}{\frac{y}{z}} \Longrightarrow \frac{w \cdot y \cdot z + x \cdot z \cdot z}{y \cdot z}$$

$$w + \frac{x}{\frac{y}{z}} \Longrightarrow w + \frac{x \cdot z}{y} \Longrightarrow \frac{w \cdot y + x \cdot z}{y}$$

So when we introduce rules to move `*` and `+` inside `/`, we insist that the the quotient involved already be a ratio of polynomials:

$$\begin{array}{lll}
\texttt{((}p_1\ \texttt{/}\ p_2\texttt{)}\ \texttt{+}\ e\texttt{)} \Longrightarrow \texttt{((}p_1\ \texttt{+}\ \texttt{(}p_2\ \texttt{*}\ e\texttt{))}\ \texttt{/}\ p_2\texttt{)} & & [\textbf{+/ distributivity}] \\
\texttt{(}e\ \texttt{+}\ \texttt{(}p_1\ \texttt{/}\ p_2\texttt{))} \Longrightarrow \texttt{(((}e\ \texttt{*}\ p_2\texttt{)}\ \texttt{+}\ p_1\texttt{)}\ \texttt{/}\ p_2\texttt{)} & & [\textbf{+/ distributivity}] \\
\texttt{((}p_1\ \texttt{/}\ p_2\texttt{)}\ \texttt{*}\ e\texttt{)} \Longrightarrow \texttt{((}p_1\ \texttt{*}\ e\texttt{)}\ \texttt{/}\ p_2\texttt{)} & & [\textbf{*/ distributivity}] \\
\texttt{(}e\ \texttt{*}\ \texttt{(}p_1\ \texttt{/}\ p_2\texttt{))} \Longrightarrow \texttt{((}e\ \texttt{*}\ p_1\texttt{)}\ \texttt{/}\ p_2\texttt{)} & & [\textbf{*/ distributivity}]
\end{array}$$

where p_1, $p_2 \in$ polynomial; $e \in$ expression

Similarly we introduce rewrite rules to combine divisions, but only if the innermost division involved is already the ratio of polynomials:[6]

$$\begin{array}{lll}
\texttt{((}p_1\ \texttt{/}\ p_2\texttt{)}\ \texttt{/}\ e\texttt{)} \Longrightarrow \texttt{(}p_1\ \texttt{/}\ \texttt{(}p_2\ \texttt{*}\ e\texttt{))} & & [\textbf{/ combination}] \\
\texttt{(}e\ \texttt{/}\ \texttt{(}p_1\ \texttt{/}\ p_2\texttt{))} \Longrightarrow \texttt{((}e\ \texttt{*}\ p_2\texttt{)}\ \texttt{/}\ p_1\texttt{)} & \text{if not } p_2 \Longrightarrow 0 & [\textbf{/ combination}] \\
\texttt{(}e\ \texttt{/}\ \texttt{(}p_1\ \texttt{/ 0))} \Longrightarrow \texttt{(0 / 0)} & & [\textbf{/ combination}]
\end{array}$$

where p_1, $p_2 \in$ polynomial; $e \in$ expression

Lastly we introduce rewrite rules to remove '`-`' operators, remove division by 1, and simplify division by 0:

[6] In this case requiring the innermost division to be a ratio of polynomials simplifies but may not be necessary for the proof of confluence in the absence of the common factor rewrite rule given below.

$$
\begin{array}{llr}
(e_1 \texttt{ - } e_2) \Longrightarrow (e_1 \texttt{ + } (\texttt{-1 * } e_2)) & & [\textbf{negation}] \\
(e \texttt{ / 1}) \Longrightarrow e & & [\textbf{divide by 1}] \\
(e \texttt{ / 0}) \Longrightarrow \texttt{(0 / 0)} & \text{if } e \not\equiv \texttt{0} & [\textbf{divide by 0}]
\end{array}
$$

where e, e_1, $e_2 \in$ expression

The total set of rewrite rules is then the rules we have introduced in this section, plus all the rewrite rules for the algebra of polynomials except the distributive rules. This set of rules can be shown to be terminating and sound. They are confluent if we restrict numbers to integers and ratios, with numeric operations giving exact results.

However, the rules are not complete. Both $\frac{5x}{5y}$ and $\frac{x}{y}$ are distinct normal forms, but we can prove they are equal using the common factors axiom. Also both $\frac{5}{2}x$ and $\frac{5x}{2}$ are distinct normal forms that we can prove equal.

To get completeness we first need to restrict the numbers to integers, and not allow ratios as numbers. This is not a limitation because any rational expression with ratio coefficients is equal to one with integer coefficients (multiply both numerator and denominator by any integer appearing in the denominator of any ratio).

Secondly we need to add the rewrite meta-rule:

$$
\begin{array}{lll}
(p_1 \texttt{ / } p_2) \Longrightarrow (p'_1 \texttt{ / } p'_2) & \text{if } (p \texttt{ * } p'_1) \Longrightarrow p_1 & [\textbf{common factors}] \\
 & \text{and } (p \texttt{ * } p'_2) \Longrightarrow p_2 & \\
 & \text{and not } p \Longrightarrow \texttt{0} & \\
 & \text{and not } p \Longrightarrow \texttt{1} &
\end{array}
$$

where p, p_1, p_2, p'_1, $p'_2 \in$ polynomial

This removes common factors from a rational expression normal form. With numbers restricted to integers, the set of all the rewrite rules, including the common factor rule, is terminating, sound, confluent, and complete.[7]

Exercise 2.4.1 Prove the following, naming the axioms used at each step:

(a) $((x \texttt{ / } y) \texttt{ * } (z \texttt{ + } w)) \texttt{ = } (((x \texttt{ * } z) \texttt{ / } y) \texttt{ + } ((w \texttt{ * } x) \texttt{ / } y))$

(b) $((x \texttt{ / } x) \texttt{ / } (y \texttt{ * } y)) \texttt{ = } (x \texttt{ / } (y \texttt{ * } (x \texttt{ * } y)))$

(c) $((x \texttt{ / } (y \texttt{ + } z)) \texttt{ + 5}) \texttt{ = } (((\texttt{5 * } z) \texttt{ + } ((\texttt{5 * } y) \texttt{ + } x)) \texttt{ / } (z \texttt{ + } y))$

[7]The requirements that quotients in various rewrite rules be ratios of polynomials can be eliminated if the common factors rule is included. In this case, confluence follows from completeness for terminating, sound rewrite rules.

Exercise 2.4.2 Rewrite to normal form in the algebra of rational expressions, naming the rewrite rules used at each step:

(a) `(((`x` / `y`) * (`x` + `y`)) + ((3 * `y`) / `x`))`
(b) `(`x` / (`y` + (`x` / (`b` + `y`))))`
(c) `((`x` * (`x` / `y`)) + ((`x` / `x`) * `y`))`

Exercise 2.4.3 In the algebra of polynomials, give all possible rewrites of the following to normal form. You need not name the rewrite rules used. To save space, you may if you like give the result as a graph with $\Longrightarrow$ arrows and nodes labeled by expressions.

(a) `((`x` / `y`) + (`w` / `z`))`
(b) `((`x` / `y`) * (`w` / `z`))`
(c) `((`x` / `y`) / (`w` / `z`))`

Exercise 2.4.4 Use the equation proof algorithm to prove or disprove the following. Show all work: in particular, show all rewrites to normal form and name the rewrite rules used at each step.

(a) `((`x` / `y`) * (`z` + `w`)) = (((`x` * `z`) / `y`) + ((`w` * `x`) / `y`))`
(b) `((`x` / `x`) / (`y` * `y`)) = (`x` / (`y` * (`x` * `y`)))`
(c) `((`x` / (`y` + `z`)) + 5) = (((5 * `x`) + ((5 * `y`) + `z`)) / (`z` + `y`))`

Exercise 2.4.5 We can use a formal algebra to compute derivatives of rational expressions. The grammar of this **algebra of derivation** is:

$$
\begin{array}{rcl}
\text{expression} & ::= & \text{number} \mid \text{variable} \\
& \mid & \texttt{(}\ \text{expression}\ \texttt{+}\ \text{expression}\ \texttt{)} \\
& \mid & \texttt{(}\ \text{expression}\ \texttt{-}\ \text{expression}\ \texttt{)} \\
& \mid & \texttt{(}\ \text{expression}\ \texttt{*}\ \text{expression}\ \texttt{)} \\
& \mid & \texttt{(}\ \text{expression}\ \texttt{/}\ \text{expression}\ \texttt{)} \\
& \mid & \texttt{(}\ \frac{d}{d\,\text{variable}}\ \text{expression}\ \texttt{)}
\end{array}
$$

Here $\frac{d}{d\,\text{variable}}$ is the derivation operator. The axioms are:

$$\begin{array}{lcll}
\frac{d}{dv}n & = & 0 & \\
\frac{d}{dv_1}v_2 & = & 0 & \text{if } v_1 \not\equiv v_2 \\
\frac{d}{dv}v & = & 1 & \\
\frac{d}{dv}\texttt{(}e_1 \texttt{ + } e_2\texttt{)} & = & \texttt{(}\frac{d}{dv}e_1 \texttt{ + } \frac{d}{dv}e_2\texttt{)} & \\
\frac{d}{dv}\texttt{(}e_1 \texttt{ - } e_2\texttt{)} & = & \texttt{(}\frac{d}{dv}e_1 \texttt{ - } \frac{d}{dv}e_2\texttt{)} & \\
\frac{d}{dv}\texttt{(}e_1 \texttt{ * } e_2\texttt{)} & = & \texttt{((}\frac{d}{dv}e_1 \texttt{ * } e_2\texttt{) + (}e_1 \texttt{ * } \frac{d}{dv}e_2\texttt{))} & \\
\frac{d}{dv}\texttt{(}e_1 \texttt{ / } e_2\texttt{)} & = & \texttt{(((}\frac{d}{dv}e_1 \texttt{ * } e_2\texttt{) - (}e_1 \texttt{ * } \frac{d}{dv}e_2\texttt{)) / (}e_2 \texttt{ * } e_2\texttt{))} &
\end{array}$$

where v, v_1, v_2 $\in$ variable; e, e_1, e_2 $\in$ expression

(a) Write rewrite rules that will reduce expressions in the algebra of derivations to normal forms that have no derivation operators.

(b) [Long] Write a LISP function that will reduce an expression in the algebra of derivations to a normal form with no derivation operators.

2.4.1 Normalizing Rational Expressions in LISP

This section continues the last section, and can be skipped or skimmed by the reader who is skipping or skimming the last section.

Just as we wrote a function `normal-3` to apply the rewrite rules of the algebra of simple expressions, we can write a function `normal` to apply the rewrite rules of the algebra of rational expressions, excluding the common factors rule, which would require a mathematical algorithm to find common factors of multi-variable polynomials.

The code for the '`apply-rules`' function that is associated with the '`normal`' is very similar to the code for the '`apply-rules-3`' function given in Figure 2.9. Some of this code is given in Figure 2.11. The only twist in coding '`apply-rules`' is to put code for rules that move `/` outside other operators before code that implements the rewrite rules of the algebra of polynomials. This ensures that the latter will only operate on expressions that are polynomials containing no `/`'s.

A test of the `normal` function that includes some of the examples given in this text appears in Figure 2.12.

```
;; Apply rules for non-polynomials first:

;; Eliminate consecutive divisions:
;;
;;   (/ e1 (/ p2 p3)) ===> (/ (* e1 p3) p2)
;;           if not p3 ===> 0
;;
((and (is/exp e)
      (is/exp (arg2 e))
      (or (not (numberp (arg2 (arg2 e))))
          (not (= (arg2 (arg2 e)) 0))))
 (rr e ...)
 (apply-rules
  `(/ ,(apply-rules `(* ,(arg1 e) ,(arg2 (arg2 e))))
      ,(arg1 (arg2 e)))))

;;   (/ e1 (/ p2 0)) ===> (/ 0 0)
;;
((and (is/exp e)
      (is/exp (arg2 e))
      (numberp (arg2 (arg2 e)))
      (/= (arg2 (arg2 e)) 0))
 (rr e '(/ 0 0))
 '(/ 0 0))

. . . . . . . . . . .

;; Apply rules for polynomials:

;; Sort terms of sums:
;;
;;   (+ m2 m1) ===> (+ m1 m2) if m1 << m2
;;
((and (is+exp e)
      (or (isfactor (arg1 e)) (is*exp (arg1 e)))
      (or (isfactor (arg2 e)) (is*exp (arg2 e)))
      (<< (arg2 e) (arg1 e)))
 (rr e `(+ ,(arg2 e) ,(arg1 e)))
 (apply-rules `(+ ,(arg2 e) ,(arg1 e))))
```

Figure 2.11: Parts of `Apply-Rules`

```
(setf *trace* t)
--->
T

(normal '(+ w (/ x (/ y z))))
--->
((/ X (/ Y Z)) => (/ (* X Z) Y))
((+ W (/ (* X Z) Y)) => (/ (+ (* W Y) (* X Z)) Y))
(/ (+ (* W Y) (* X Z)) Y)

(normal '(* (/ (* x x) (+ x y)) (+ x y)))
--->
((* (/ (* X X) (+ X Y)) (+ X Y)) =>
  (/ (* (* X X) (+ X Y)) (+ X Y))
)
((* (* X X) (+ X Y)) => (* X (* X (+ X Y))))
((* X (+ X Y)) => (+ (* X X) (* X Y)))
((* X (+ (* X X) (* X Y))) =>
  (+ (* X (* X X)) (* X (* X Y)))
)
(/ (+ (* X (* X X)) (* X (* X Y))) (+ X Y))

(setf *trace* nil)
--->
NIL

(normal
  '(+ -4 (+ (* c 4) (+ 4 (+ (* b 9)
                            (+ (* -4 c)
                               (+ d (* b -8))))))))
--->
(+ B D)

(normal '(+ 1 (/ 1 (+ 1 (/ 1 (+ 1 (/ 1 (+ 1 x))))))))
--->
(/ (+ 5 (* 3 X)) (+ 3 (* 2 X)))
```

Figure 2.12: Normal Test

2.5 Propositional Calculus

We have now finished our study of rational arithmetic expressions, and will take up the study of corresponding problems in the area of boolean (true/false) logic. First we introduce 'boolean formulae', which are built from two constants, 'true' and 'false', and the logical operators 'or', 'and', and 'not'. When we add variables that can take true or false as values, we get a subject called 'propositional calculus', that we will study extensively. Formula in propositional calculus have normal forms that can be useful in proving propositional theorems.

2.5.1 Boolean Formulae

Boolean formulae[8] express computation with the constants 'true' and 'false' and the logical operators 'or', 'and', and 'not'. Boolean formulae are defined by the grammar:

$$
\begin{array}{lcl}
\text{boolean-formula} & ::= & \text{constant} \\
 & | & (\ \text{boolean-formula} \lor \text{boolean-formula}\ ) \\
 & | & (\ \text{boolean-formula} \land \text{boolean-formula}\ ) \\
 & | & (\ \neg\ \text{boolean-formula}\ ) \\
\text{constant} & ::= & \mathbf{f} \mid \mathbf{t}
\end{array}
$$

The constants **t**, **f** denote **true** and **false** respectively. These two constants are also called '**boolean values**'.

The **logical operators** $\land$, $\lor$, and $\neg$ denote logical '**and**', '**or**', and '**not**' respectively.

Just as arithmetic may be defined by addition and multiplication tables, the logical operators may be defined by tables. However, because there are only two constant values in boolean formulae, the operations can be completely defined by one small table:

p	q	$p \lor q$	$p \land q$	$\neg p$
f	**f**	**f**	**f**	**t**
f	**t**	**t**	**f**	
t	**f**	**t**	**f**	**f**
t	**t**	**t**	**t**	

[8] Named after the mathematician George Boole.

A table in this form consisting of the input variable columns and one output expression column is called the '**truth table**' of the output expression. Thus the first three columns of the table just given form the truth table of $p \vee q$, and the first two columns plus the fourth column form the truth table of $p \wedge q$. A truth table gives the boolean value of the output expression as a function of the boolean values of the input variables.

An example of a boolean computation is:

$$(\mathbf{t} \wedge \mathbf{f}) \Longrightarrow \mathbf{f} \qquad (\neg\ \mathbf{t}) \Longrightarrow \mathbf{f}$$

$$(\mathbf{t} \vee (\mathbf{t} \wedge (\neg\ \mathbf{t}))) \Longrightarrow (\mathbf{t} \vee (\mathbf{t} \wedge \mathbf{f})) \Longrightarrow (\mathbf{t} \vee \mathbf{f}) \Longrightarrow \mathbf{t}$$

Given everything we have just said, we can introduce the formal **algebra of boolean formulae**, whose expressions are boolean formulae, whose axioms are:

$$\begin{array}{lll}
(\neg\ \mathbf{f}) = \mathbf{t} & (\mathbf{f} \vee \mathbf{f}) = \mathbf{f} & (\mathbf{f} \wedge \mathbf{f}) = \mathbf{f} \\
(\neg\ \mathbf{t}) = \mathbf{f} & (\mathbf{f} \vee \mathbf{t}) = \mathbf{t} & (\mathbf{f} \wedge \mathbf{t}) = \mathbf{f} \\
 & (\mathbf{t} \vee \mathbf{f}) = \mathbf{t} & (\mathbf{t} \wedge \mathbf{f}) = \mathbf{f} \\
 & (\mathbf{t} \vee \mathbf{t}) = \mathbf{t} & (\mathbf{t} \wedge \mathbf{t}) = \mathbf{t}
\end{array}$$

and whose rewrite rules are:

$$\begin{array}{lll}
(\neg\ \mathbf{f}) \Longrightarrow \mathbf{t} & (\mathbf{f} \vee \mathbf{f}) \Longrightarrow \mathbf{f} & (\mathbf{f} \wedge \mathbf{f}) \Longrightarrow \mathbf{f} \\
(\neg\ \mathbf{t}) \Longrightarrow \mathbf{f} & (\mathbf{f} \vee \mathbf{t}) \Longrightarrow \mathbf{t} & (\mathbf{f} \wedge \mathbf{t}) \Longrightarrow \mathbf{f} \\
 & (\mathbf{t} \vee \mathbf{f}) \Longrightarrow \mathbf{t} & (\mathbf{t} \wedge \mathbf{f}) \Longrightarrow \mathbf{f} \\
 & (\mathbf{t} \vee \mathbf{t}) \Longrightarrow \mathbf{t} & (\mathbf{t} \wedge \mathbf{t}) \Longrightarrow \mathbf{t}
\end{array}$$

The normal forms of boolean formulae are just the constants **f** and **t**, and rewriting a boolean formulae to a normal form is the same as evaluating the formulae.

Exercise 2.5.1.1 Rewrite to normal form in the algebra of boolean formulae:

(a) $(\mathbf{t} \wedge ((\mathbf{f} \vee (\neg\ \mathbf{f})) \wedge \mathbf{f}))$

(b) $((\mathbf{f} \wedge (\mathbf{t} \vee \mathbf{t})) \vee (\neg\ \mathbf{t}))$

(c) $((\mathbf{f} \vee (\neg\ \mathbf{f})) \wedge (\mathbf{t} \vee (\neg\ \mathbf{t})))$

2.5.2 Well Formed Formulae

We now want to include variables in our formulae. We get the formal **algebra of propositional calculus with constants**, whose expressions are called **well-formed-formulae**, or **wffs**. The grammar is:

$$
\begin{array}{lcl}
\text{wff} & ::= & \text{constant} \mid \text{variable} \\
 & \mid & (\ \text{wff} \vee \text{wff}\ ) \\
 & \mid & (\ \text{wff} \wedge \text{wff}\ ) \\
 & \mid & (\ \neg\text{wff}\ ) \\
\text{constant} & ::= & \mathbf{f} \mid \mathbf{t} \\
\text{variable} & ::= & x \mid y \mid z \mid x_1 \mid y_1 \mid z_1 \mid \ldots
\end{array}
$$

The motive behind the name 'well-formed formulae' is to emphasize that formulae such as $(x\neg y)$ that are <u>not</u> well formed need not be considered.

Some examples of wff are:

$$(x \wedge y) \qquad (x \wedge (y \wedge z)) \qquad ((x \wedge y) \wedge z)$$

$$(x \wedge (y \vee (\neg (x_1 \vee x_3)))) \qquad ((\mathbf{f} \vee (x \wedge \mathbf{t})) \vee (y \wedge (\neg x)))$$

Variables in wff denote '**propositions**' like 'Bob walked Sparky this morning' that take the values **t** and **f**.

If we exclude constants from well-formed formulae, we get the **algebra of propositional calculus**. We will shortly see that every wff with constants is provably equal to a simpler wff with no constants,[9] so in this sense constants are unnecessary in wff, and mathematicians are used to excluding constants from the formulae of propositional calculus. However, we will find it convenient to compute truth tables of wff by substituting constants for variables, so we permit constants in our wff.

2.5.3 Logical Equivalence

An '**assignment**' is a mapping from variables to constants: e.g. $x_1 \mapsto \mathbf{t}$, $x_2 \mapsto \mathbf{f}$, $y \mapsto \mathbf{f}$, ...

[9] With the 'trivial' exception of wff that are provably equal to either **f** or **t**.

For two wff w_1 and w_2, $w_1 \equiv_\ell w_2$ means that for any assignment that assigns **t** or **f** to every variable in w_1 or w_2, then w_1 and w_2 have the same value as boolean formulae after replacing each variable by its assigned constant value. In this case we say that w_1 and w_2 are '**logically equivalent**'.

Some examples of logical equivalences are:

$$(x \lor y) \equiv_\ell (y \lor x) \qquad\qquad (\neg\ (\neg\ x_1)) \equiv_\ell x_1$$

Lemma: $w_1 \equiv_\ell w_2$ if and only if w_1 and w_2 have the same truth table.

Thus $\qquad (x \lor (y \land z)) \equiv_\ell ((x \lor y) \land (x \lor z))$

can be deduced from:

x	y	z	$(y \land z)$	$(x \lor (y \land z))$	$(x \lor y)$	$(x \lor z)$	$((x \lor y) \land (x \lor z))$
f	**f**	**f**	**f**	**f**	**f**	**f**	**f**
f	**f**	**t**	**f**	**f**	**f**	**t**	**f**
f	**t**	**f**	**f**	**f**	**t**	**f**	**f**
f	**t**	**t**	**t**	**t**	**t**	**t**	**t**
t	**f**	**f**	**f**	**t**	**t**	**t**	**t**
t	**f**	**t**	**f**	**t**	**t**	**t**	**t**
t	**t**	**f**	**f**	**t**	**t**	**t**	**t**
t	**t**	**t**	**t**	**t**	**t**	**t**	**t**

The symbol '$\equiv$' is usually used to mean logical equivalence, but we use the symbol '$\equiv_\ell$' instead, for we are already using '$\equiv$' to mean syntactic identity in this chapter.

2.5.4 Standard Equivalences

There are a number of standard equivalences in propositional calculus that are often used to compute with wff:[10]

[10] Any algebraic system with operators $\lor$, $\land$, and $\neg$ that obeys all of these equivalences that contain no constants is called a **boolean algebra**. Another example of a boolean algebra is set theory with the operators interpreted as union, intersection, and complement.

$(x \wedge y) \equiv_\ell (y \wedge x)$	$(x \vee y) \equiv_\ell (y \vee x)$	[**commutativity**]
$(x \wedge (y \wedge z)) \equiv_\ell ((x \wedge y) \wedge z)$ $(x \vee (y \vee z)) \equiv_\ell ((x \vee y) \vee z)$		[**associativity**]
$(x \wedge x) \equiv_\ell x$	$(x \vee x) \equiv_\ell x$	[**idempotency**]
$(\neg (\neg x)) \equiv_\ell x$		[**double negation**]
$(x \wedge (y \vee z)) \equiv_\ell ((x \wedge y) \vee (x \wedge z))$ $(x \vee (y \wedge z)) \equiv_\ell ((x \vee y) \wedge (x \vee z))$		[**distributivity**]
$(\neg (x \wedge y)) \equiv_\ell ((\neg x) \vee (\neg y))$ $(\neg (x \vee y)) \equiv_\ell ((\neg x) \wedge (\neg y))$		[**DeMorgan's laws**]
$(\mathbf{t} \vee x) \equiv_\ell \mathbf{t}$	$(\mathbf{f} \wedge x) \equiv_\ell \mathbf{f}$	[**nullity**]
$(\mathbf{f} \vee x) \equiv_\ell x$	$(\mathbf{t} \wedge x) \equiv_\ell x$	[**unity**]
$(x \vee (\neg x)) \equiv_\ell \mathbf{t}$	$(x \wedge (\neg x)) \equiv_\ell \mathbf{f}$	[**excluded middle**]

For the axioms of the formal algebra of propositional calculus with constants we take the equivalences just given with '$\equiv_\ell$' replaced by '='.

Lemma: for wff w_1 and w_2, $w_1 = w_2$ is provable if and only if $w_1 \equiv_\ell w_2$.

Proof Comments: Showing that if $w_1 = w_2$ is provable then $w_1 \equiv_\ell w_2$ is straight forward, because the axioms are known logical equivalences, and the other proof rules are sound for logical equivalence. Showing the converse, that if $w_1 \equiv_\ell w_2$ then $w_1 = w_2$ is provable, is harder.[11] QED

Therefore the axioms are 'complete' in the sense that all logical equivalences can be proved from the axioms.

Exercise 2.5.4.1 Prove the following, naming the axioms used at each step:

(a) $((x \vee (\neg z)) \wedge (z \vee (\neg x))) = ((x \wedge z) \vee ((\neg z) \wedge (\neg x)))$
(b) $(p \vee (\neg q)) = (\neg (q \wedge (\neg p)))$
(c) $(\neg (p \vee (\neg (q \vee r)))) = (((\neg p) \wedge q) \vee (r \wedge (\neg p)))$

[11] We can show that any wff w is provably equal to a disjunctive normal form (see below) in which every clause contains each variable in w in exactly one literal, no literal is duplicated in a clause, and no two clauses contain the same literals. Then we can prove that two disjunctive normal forms of this kind are logically equivalent if and only if they are provably equal.

2.5.5 Rewrite Rules for Propositional Calculus

There are several different sets of rewrite rules that are useful with the algebras of propositional calculus.

Constant eliminating rewrite rules that can be used to remove constants from wff are:

$$
\begin{array}{lll}
(\mathbf{t} \lor w) \Longrightarrow \mathbf{t} & (w \lor \mathbf{t}) \Longrightarrow \mathbf{t} & [\textbf{nullity}] \\
(\mathbf{f} \land w) \Longrightarrow \mathbf{f} & (w \land \mathbf{f}) \Longrightarrow \mathbf{f} & \\
(\mathbf{f} \lor w) \Longrightarrow w & (w \lor \mathbf{f}) \Longrightarrow w & [\textbf{unity}] \\
(\mathbf{t} \land w) \Longrightarrow w & (w \land \mathbf{t}) \Longrightarrow w & \\
(\neg\ \mathbf{f}) \Longrightarrow \mathbf{t} & (\neg\ \mathbf{t}) \Longrightarrow \mathbf{f} & [\textbf{negation}]
\end{array}
$$

where $w \in \text{wff}$

The normal forms of these rules are wff with no constants plus the two wffs '**f**' and '**t**' consisting of constants by themselves.

These rules are terminating (each removes one constant), sound, and confluent. Of course they are not complete: $(x \land (\neg\ x))$ and $(y \land (\neg\ y))$ are provably equal but distinct normal forms.

The distributivity axioms can be turned into the rewrite rules such as:

$$
\begin{array}{lll}
((w_1 \lor w_2) \land w_3) \Longrightarrow ((w_1 \land w_3) \lor (w_2 \land w_3)) & & [\land\lor\textbf{distributivity}] \\
(w_1 \land (w_2 \lor w_3)) \Longrightarrow ((w_1 \land w_2) \lor (w_1 \land w_3)) & & \\
((w_1 \land w_2) \lor w_3) \Longrightarrow ((w_1 \lor w_3) \land (w_2 \lor w_3)) & & [\lor\land\textbf{distributivity}] \\
(w_1 \lor (w_2 \land w_3)) \Longrightarrow ((w_1 \lor w_2) \land (w_1 \lor w_3)) & &
\end{array}
$$

where $w_1,\ w_2,\ w_3 \in \text{wff}$

However, if both $\land\lor$ and $\lor\land$ distributivity rewrite rules are included in the same set of rewrite rules, that set is non terminating, as is illustrated by:

$$
\begin{array}{lll}
(x \land (y \lor z)) & \Longrightarrow & ((x \land y) \lor (x \land z)) \\
& \Longrightarrow & (((x \land y) \lor x) \land ((x \land y) \lor z)) \\
& \Longrightarrow & \ldots
\end{array}
$$

in which the wffs keep growing and growing in size. This is not surprising as the $\wedge\vee$ rule is useful for moving $\wedge$'s inside $\vee$'s and the $\vee\wedge$ rule is useful for doing the reverse. We should not want to do both.

So we have a choice. If we move $\wedge$'s inside $\vee$'s the normal forms that result are called 'disjunctive normal forms'. If instead we move $\vee$'s inside $\wedge$'s the normal forms that result are called 'conjunctive normal forms'. In both cases we use DeMorgan's law rules to move $\neg$'s inside both $\wedge$'s and $\vee$'s, and a double negation rule to eliminate double negations.

A grammar describing **disjunctive normal forms**, also known as **dnfs**, is:

$$
\begin{array}{lcl}
\text{dnf} & ::= & \text{dnf-clause} \\
 & | & (\ \text{dnf} \vee \text{dnf}\ ) \\
\text{dnf-clause} & ::= & \text{literal} \\
 & | & (\ \text{dnf-clause} \wedge \text{dnf-clause}\ ) \\
\text{literal} & ::= & \text{variable} \mid (\ \neg\ \text{variable}\ )
\end{array}
$$

Thus a **literal** is a variable or the negation of a variable, a **dnf-clause** is the $\wedge$ of literals, and a **dnf** is the $\vee$ of dnf-clauses. Dnfs have $\wedge$'s inside $\vee$'s, $\neg$'s inside $\wedge$'s and $\vee$'s, and no double negations. Notice that we have excluded constants from dnfs; this is because the theory of dnfs belongs to propositional calculus proper, without constants, and we have no use for constants in dnfs.

Dnf rewrite rules that can be used to convert wffs without constants into dnfs are:

$$
\begin{array}{ll}
((w_1 \ \vee \ w_2) \ \wedge \ w_3) \implies ((w_1 \ \wedge \ w_3) \ \vee \ (w_2 \ \wedge \ w_3)) & [\wedge\vee\textbf{distributivity}] \\
(w_1 \ \wedge \ (w_2 \ \vee \ w_3)) \implies ((w_1 \ \wedge \ w_2) \ \vee \ (w_1 \ \wedge \ w_3)) & \\
(\neg \ (w_1 \ \wedge \ w_2)) \implies ((\neg \ w_1) \ \vee \ (\neg \ w_2)) & [\textbf{DeMorgan's law}] \\
(\neg \ (w_1 \ \vee \ w_2)) \implies ((\neg \ w_1) \ \wedge \ (\neg \ w_2)) & \\
(\neg \ (\neg w)) \implies w & [\textbf{double negation}]
\end{array}
$$

where w, w_1, w_2, $w_3 \in$ wff

An example using these rules to rewrite a wff to a dnf is:

$$
\begin{array}{ll}
 & \texttt{(}\neg\ \texttt{(}p \lor \texttt{(}q \land \texttt{(}\neg\ p\texttt{))))} \\
\Longrightarrow & \texttt{((}\neg\ p\texttt{)} \land \texttt{(}\neg\ \texttt{(}q \land \texttt{(}\neg\ p\texttt{))))} \\
\Longrightarrow & \texttt{((}\neg\ p\texttt{)} \land \texttt{((}\neg\ q\texttt{)} \lor \texttt{(}\neg\ \texttt{(}\neg\ p\texttt{))))} \\
\Longrightarrow & \texttt{((}\neg\ p\texttt{)} \land \texttt{((}\neg\ q\texttt{)} \lor p\texttt{))} \\
\Longrightarrow & \texttt{(((}\neg\ p\texttt{)} \land \texttt{(}\neg\ q\texttt{))} \lor \texttt{((}\neg\ p\texttt{)} \land p\texttt{))}
\end{array}
$$

This set of rules is terminating and sound. It is not confluent, but would become so if rules were added to sort literals in a dnf-clause and dnf-clauses in a dnf. Even with the addition of sorting rules, the set of rules would not be complete: $(x \land (\neg\ x))$ and $(y \land (\neg\ y))$ are provably equal but distinct dnf.

The theory of **conjunctive normal forms**, also known as **cnfs**, is the same as that of dnfs, but with $\land$ and $\lor$ interchanged. The grammar of cnfs is:

$$
\begin{array}{lcl}
\text{cnf} & ::= & \text{cnf-clause} \\
 & \mid & (\ \text{cnf} \land \text{cnf}\ ) \\
\text{cnf-clause} & ::= & \text{literal} \\
 & \mid & (\ \text{cnf-clause} \lor \text{cnf-clause}\ ) \\
\text{literal} & ::= & \text{variable} \mid (\ \neg\ \text{variable}\ )
\end{array}
$$

Cnf rewrite rules that can be used to convert wffs without constants into cnfs are:

$$
\begin{array}{ll}
((w_1 \land w_2) \lor w_3) \Longrightarrow ((w_1 \lor w_3) \land (w_2 \lor w_3)) & [\lor\land\textbf{distributivity}] \\
(w_1 \lor (w_2 \land w_3)) \Longrightarrow ((w_1 \lor w_2) \land (w_1 \lor w_3)) & \\
(\neg\ (w_1 \land w_2)) \Longrightarrow ((\neg\ w_1) \lor (\neg\ w_2)) & [\textbf{DeMorgan's law}] \\
(\neg\ (w_1 \lor w_2)) \Longrightarrow ((\neg\ w_1) \land (\neg\ w_2)) & \\
(\neg\ (\neg w)) \Longrightarrow w & [\textbf{double negation}]
\end{array}
$$

where w, w_1, w_2, $w_3 \in$ wff

Exercise 2.5.5.1 Rewrite to normal form using the constant elimination rewrite rules (but do not use any other rules):

(a) $((x \land (\mathbf{t} \lor (\neg\ y))) \land \mathbf{t})$
(b) $((\mathbf{f} \lor (\mathbf{t} \land z)) \lor (\neg\ \mathbf{f}))$
(c) $(\mathbf{f} \lor ((\neg\ x) \land (y \lor (\neg\ \mathbf{t}))))$

Exercise 2.5.5.2 Rewrite to dnf using the dnf rewrite rules (but do not use any other rules):

(a) $((p \lor q) \lor (p \land (\neg q)))$
(b) $((p \land q) \land (p \lor (\neg q)))$
(c) $((\neg (p \lor q)) \lor (p \land (\neg q)))$
(d) $((\neg (p \land q)) \land (p \lor (\neg q)))$
(e) $((((\neg q) \land p) \lor ((\neg p) \land r)) \lor ((\neg r) \land s))$
(f) $((((\neg q) \lor p) \land ((\neg p) \lor r)) \land ((\neg r) \lor s))$
(g) $(\neg ((((\neg q) \land p) \lor ((\neg p) \land r)) \lor ((\neg r) \land s)))$
(h) $(\neg ((((\neg q) \lor p) \land ((\neg p) \lor r)) \land ((\neg r) \lor s)))$

Exercise 2.5.5.3 Rewrite to cnf using the cnf rewrite rules (but do not use any other rules):

(a) $((p \lor q) \lor (p \land (\neg q)))$
(b) $((p \land q) \land (p \lor (\neg q)))$
(c) $((\neg (p \lor q)) \lor (p \land (\neg q)))$
(d) $((\neg (p \land q)) \land (p \lor (\neg q)))$
(e) $((((\neg q) \land p) \lor ((\neg p) \land r)) \lor ((\neg r) \land s))$
(f) $((((\neg q) \lor p) \land ((\neg p) \lor r)) \land ((\neg r) \lor s))$
(g) $(\neg ((((\neg q) \land p) \lor ((\neg p) \land r)) \lor ((\neg r) \land s)))$
(h) $(\neg ((((\neg q) \lor p) \land ((\neg p) \lor r)) \land ((\neg r) \lor s)))$

2.5.6 Theorems and Unsatisfiability

If $w \in$ wff, w is a **theorem** if and only if $w \equiv_\ell \mathbf{t}$.

If $w \in$ wff, w is **unsatisfiable** if and only if $w \equiv_\ell \mathbf{f}$.

Lemma: If $w \in$ wff, w is a theorem if and only if $(\neg w)$ is unsatisfiable.

This leads to the first of three methods we will introduce to use a computer to prove that a wff is a theorem. Such methods are called '**automatic theorem proving**' algorithms.

Automatic Theorem Proving Algorithm 1: To decide if a wff w is a theorem, make a truth table for w, and see if all the lines in the table contain $\mathbf{t}$.

This algorithm has a big drawback: if there are 100 variables in w there are $2^{100} \approx 10^{30}$ rows in the truth table. So we look for other automatic theorem proving algorithms.

Another method is based on the observation that it is easy to check whether a dnf is satisfiable. Clearly it can be satisfied if any of its dnf clauses can be satisfied. A clause can be satisfied unless it contains both the literals v and $(\neg\ v)$ for some variable v. Therefore:

Automatic Theorem Proving Algorithm 2: To decide if a wff w is a theorem:

(1) Rewrite $(\neg\ w)$ to dnf using the dnf rewrite rules.

(2) Check that <u>all</u> the dnf-clauses are unsatisfiable.
If yes, $(\neg\ w)$ is unsatisfiable and w is a theorem.
If no, w is not a theorem.

(2a) A dnf-clause if unsatisfiable if and only if it contains both v and $(\neg\ v)$ for some variable v.

As an example of how (2a) is applied, consider the dnf $(((\neg p)\wedge p)\vee((\neg q)\wedge p))$. The first clause is unsatisfiable but the second clause can be satisfied by the assignment $q \mapsto \mathbf{f}$, $p \mapsto \mathbf{t}$.

Unfortunately the time complexity for rewriting wff to dnf can be $O(2^N)$, where N is the length of the input wff. Because the worst case tends to be approached in practical applications, automatic theorem proving algorithm 2 is not that useful. A more useful third algorithm, called resolution theorem proving, uses cnf instead of dnf, but is harder to describe and justify.

Exercise 2.5.6.1 Construct a complete truth table for each of the following wff, showing also the truth tables of important subexpressions of the wff. Then, using the method of automatic theorem proving algorithm 1, determine if the wff is a theorem.

(a) $((\neg\ ((p\ \vee\ (\neg\ q))\ \wedge\ q))\ \vee\ p)$
(b) $((\neg\ (((\neg\ p)\ \vee\ (\neg\ q))\ \wedge\ q))\ \vee\ p)$
(c) $((p\ \vee\ q)\ \vee\ ((\neg\ p)\ \wedge\ (\neg\ q)))$
(d) $((p\ \vee\ q)\ \vee\ (p\ \wedge\ (\neg\ q)))$
(e) $((\neg\ (((\neg\ p)\ \vee\ \mathtt{q})\ \wedge\ (((\neg\ q)\ \vee\ r)\ \wedge\ p)))\ \vee\ r)$
(f) $((p\ \wedge\ q)\ \vee\ ((p\ \wedge\ r)\ \vee\ ((r\ \wedge\ q)\ \vee$
$(((\neg\ p)\ \wedge\ (\neg\ q))\ \vee$
$(((\neg\ p)\ \wedge\ (\neg\ r))\ \vee\ ((\neg\ q)\ \wedge\ (\neg\ r)))))))$

Exercise 2.5.6.2 Apply the method of automatic theorem proving algorithm 2 to decide whether the following wff are theorems. Carefully show rewrites using the dnf rewrite rules.

(a) $((\neg\ ((p \lor (\neg\ q)) \land q)) \lor p)$
(b) $((\neg\ (((\neg\ p) \lor (\neg\ q)) \land q)) \lor p)$
(c) $((p \lor q) \lor ((\neg\ p) \land (\neg\ q)))$
(d) $((p \lor q) \lor (p \land (\neg\ q)))$
(e) $((\neg\ (((\neg\ p) \lor \mathtt{q}) \land (((\neg\ q) \lor r) \land p))) \lor r)$
(f) $((p \land q) \lor ((p \land r) \lor ((r \land q) \lor$
$(((\neg\ p) \land (\neg\ q)) \lor$
$(((\neg\ p) \land (\neg\ r)) \lor ((\neg\ q) \land (\neg\ r)))))))$

2.5.7 Propositional Calculus in LISP

In order to write programs in LISP that perform propositional calculations, we need to agree on what LISP s-expressions will be used to represent wff. This can be done by giving the grammar:

wff	::=	constant | variable
	|	`(or` wff wff `)`
	|	`(and` wff wff `)`
	|	`(not` wff `)`
constant	::=	`f` | `t`
variable	::=	symbol except constant or `nil`
assignment	::=	`()`
	|	`( (` variable wff `) .` assignment `)`

An assignment is a representation of a list of variable-wff pairs. Each pair means that the variable is to be replaced by the wff when substitution is performed using the assignment. Thus

```
((u (and p q)) (v t) (w (not q)))
```

means the assignment `u`$\mapsto$`(and p q)`, `v`$\mapsto$`t`, `w`$\mapsto$`(not q)`.

The following definition is often useful: if w_1 and w_2 are wff,

$$\texttt{(implies } w_1\ w_2\texttt{)} \equiv \texttt{(or (not } w_1\texttt{) } w_2\texttt{)}$$

Here $\equiv$ means 'equivalent on input' in the sense of LISP, but it could just as well mean 'logically equivalent' in the sense of propositional calculus.

Below we give code for some of the following LISP functions. The others are left to the reader to write as exercises:

`(evaluate '`b`)`	Evaluate the boolean formula b.
`(wff? '`β`)`	Return true iff β is a wff.
`(assign '`α` '`w`)`	Substitute using assignment α into wff w.
`(dnf '`w`)`	Return dnf of a wff w with no constants.
`(literals '`w`)`	Return list of literals in a wff w with no constants.
`(variables '`w`)`	Return list of variables in a wff w.
`(theorem? '`w`)`	Return true if a wff w is a theorem. (But what algorithm should be use?)

where $b \in$ boolean-formulae; $w \in$ wff; $\alpha \in$ assignment; $\beta \in$ s-expression

Note that we would prefer the name `substitute` for the `assign` function, but the there is already a function named `substitute` in COMMONLISP.

Some example applications of the above functions are:

```
(evaluate '(and t (not f))) ⟹ 't

(assign '((x t) (y (or p q))) '(and x (not y)))
  ⟹ '(and t (not (or p q)))
```

Code for the `wff?` and `assign` functions is given in Figures 2.13, 2.14, and 2.15.

Assignments have the form of a list of sublists with the first element of each sublist being a **key** that is to be looked up. Such a data structure is common in LISP, and is called an **association list**. There is a LISP function, `assoc`, designed to look up a sublist beginning with an atom, and return that sublist, or return `nil` if there is no such sublist. Formally:

```
(assoc 'α '()) ⟹ nil
(assoc 'α '((β . γ) . δ))  ⟹ '(β . γ)          if (eql 'α 'β)
                           ⟹ '(assoc 'α 'δ)   otherwise
```

where $\alpha \in$ atom, β, γ, $\delta \in$ s-expression

```
;;;; Propositional Calculus
;;;;
;;;; File:         propositional-calculus.lsp
;;;; Author:       <your name> (<your e-mail address>)
;;;; Version:      1

;; In the following it is assumed that the symbols T,
;; F, and NIL are NEVER used as variables.
;;
;; It is also assumed that DNF formula do NOT contain
;; the constants T or F.
;;
;; The IMPLIES operation is NOT handled by this code.
;;
;; Most of the functions below are defined only on
;; input obeying particular assumptions.  None of these
;; functions makes any attempt to check for inputs that
;; do not obey the required assumptions.  E.g. many
;; functions work only on wff, and some only on DNF wff
;; or DNF clauses.


;; Functions to make code more readable.
;;
(defun op (e) (car e))
(defun arg1 (e) (cadr e))
(defun arg2 (e) (caddr e))
;;
(defun is-not-exp (e) (and (consp e) (eql 'not (op e))))
(defun is-and-exp (e) (and (consp e) (eql 'and (op e))))
(defun is-or-exp (e) (and (consp e) (eql 'or (op e))))
```

Figure 2.13: Propositional Calculus Functions: Page 1 of 3

```
;; Determine whether an s-expression e is a well formed
;; formula of the propositional calculus with constants.
;; Both constants and variables may appear in e, but NIL
;; may not.  Assumes e does NOT contain any dotted
;; sublists.
;;
;; RR: (wff? 's) ===> (not (eql 'nil 's))  s is a symbol
;;     (wff? `(not e1)) ===> (wff? e1)
;;     (wff? `(and e1 e2)) ===>
;;         (and (wff? (arg1 e)) (wff? (arg2 e)))
;;     (wff? `(or e1 e2)) ===>
;;         (and (wff? (arg1 e)) (wff? (arg2 e)))
;;     (wff? `e) ===> false     if above not applicable
;;
(defun wff? (e)
  (cond
   ((symbolp e)
    (not (eql 'nil e)))
   ((and (is-not-exp e) (= 2 (length e)))
    (wff? (arg1 e)))
   ((and (is-and-exp e) (= 3 (length e)))
    (and (wff? (arg1 e))
         (wff? (arg2 e))))
   ((and (is-or-exp e) (= 3 (length e)))
    (and (wff? (arg1 e))
         (wff? (arg2 e))))))
```

Figure 2.14: Propositional Calculus Functions: Page 2 of 3

```
;; Substitute in a wff e for the variables in an
;; assignment a.  The name ASSIGN is chosen for
;; this function because COMMONLISP already has
;; a function named SUBSTITUTE.
;;
;; RR: (assign 'a 't) ===> 't
;;     (assign 'a 'f) ===> 'f
;;     (assign 'a 'v)
;;         ===> (second (assoc 'v 'a))  if this is not
;;                                              nil
;;         ===> 'v                      otherwise
;;     (assign 'a '(not e))
;;         ===> `(not ,(assign 'a 'e))
;;     (assign 'a '(and e1 e2))
;;         ===> `(and ,(assign 'a 'e1)
;;                    ,(assign 'a 'e2))
;;     (assign 'a '(or e1 e2))
;;         ===> `(or ,(assign 'a 'e1)
;;                   ,(assign 'a 'e2))
;;
(defun assign (a e)
  (cond
   ((eql 't e)
    't)
   ((eql 'f e)
    'f)
   ((symbolp e)
    (or (second (assoc e a)) e))
   ((is-not-exp e)
    `(not ,(assign a (arg1 e))))
   ((is-and-exp e)
    `(and ,(assign a (arg1 e))
          ,(assign a (arg2 e))))
   ((is-or-exp e)
    `(or ,(assign a (arg1 e))
         ,(assign a (arg2 e))))))
```

Figure 2.15: Propositional Calculus Functions: Page 3 of 3

Exercise 2.5.7.1 Give rewrite rule definitions (not `defun`'s) for the following functions (some of which are described above in this section):

(a) `evaluate`.
(b) `variables`.
(c) `theorem?-1` that uses automatic theorem proving algorithm 1 and the `evaluate`, `variables`, and `assign` functions.
(d) `literals`.
(e) `dnf`. Use the dnf rewrite rules. Write a helper function `dnf-distribute` that plays the role of `apply-rules` for expressions with an '`and`' or '`or`' operator and normal operands. Handle '`not`' in the '`dnf`' function without putting its operand into normal form first.
(f) `negate`, that negates a literal.
(g) `unsatisfiable-dnf-clause`, that returns true if a dnf clause is unsatisfiable. Use `negate` and `member`.
(h) `unsatisfiable-dnf`, that returns true if a dnf is unsatisfiable. Use `unsatisfiable-dnf-clause`.
(i) `theorem?-2`, that uses automatic theorem proving algorithm 2 and the `dnf` and `unsatisfiable-dnf` functions.

Exercise 2.5.7.2 Give the function definitions (i.e., `defun`'s) for the following functions from the last exercise:

(a) `evaluate`
(b) `variables`
(c) `theorem?-1`
(d) `literals`
(e) `dnf`
(f) `negate`
(g) `unsatisfiable-dnf-clause`
(h) `unsatisfiable-dnf`
(i) `theorem?-2`

2.6 Proofs in Symbolic Algebra

In this section we will discuss termination, soundness, confluence, and completeness of sets of rewrite rules with the kind of precision one would expect of a mathematician. We include some proofs that rewrite systems introduced in this chapter have various of these properties. The purpose of this section is to clarify these concepts and give proofs involving them, and to demonstrate the utility of pure mathematical methods in computer science.

If the reader looks in the literature for the topics of this chapter, he will find these topics (e.g., in [Mit96]), but with different terminology. The different terminology arises from a whole different point of view that has historical origins in mathematics and not computer science, and because of this different view, the terminology differences are not trivially reconcilable. This is discussed at the end of this section.

This section can be skipped or skimmed by the reader, and no other part of the book depends upon it.

2.6.1 Rewrite Relations

In the rest of this book we 'abuse language' by taking $\Longrightarrow$ to mean 'rewrites to in one or more steps'. Thus for the algebra of simple expressions we might write:

$$
\begin{array}{lll}
\texttt{(}z\ \texttt{+}\ \texttt{(}y\ \texttt{+}\ x\texttt{))} \Longrightarrow \texttt{(}y\ \texttt{+}\ \texttt{(}z\ \texttt{+}\ x\texttt{))} & & \text{[One Step]} \\
\texttt{(}y\ \texttt{+}\ \texttt{(}z\ \texttt{+}\ x\texttt{))} \Longrightarrow \texttt{(}y\ \texttt{+}\ \texttt{(}x\ \texttt{+}\ z\texttt{))} & & \text{[One Step]} \\
\texttt{(}z\ \texttt{+}\ \texttt{(}y\ \texttt{+}\ x\texttt{))} \Longrightarrow \texttt{(}y\ \texttt{+}\ \texttt{(}x\ \texttt{+}\ z\texttt{))} & & \text{[Two Steps]}
\end{array}
$$

But in this section we will be more precise: $\Longrightarrow$ means 'rewrites in exactly one step' and $\Longrightarrow^\star$ means 'rewrites in zero or more steps'. Thus in the algebra of simple expressions:

$$
\begin{array}{lll}
\texttt{(}z\ \texttt{+}\ \texttt{(}y\ \texttt{+}\ x\texttt{))} \Longrightarrow \texttt{(}y\ \texttt{+}\ \texttt{(}z\ \texttt{+}\ x\texttt{))} & & \text{[One Step]} \\
\texttt{(}y\ \texttt{+}\ \texttt{(}z\ \texttt{+}\ x\texttt{))} \Longrightarrow \texttt{(}y\ \texttt{+}\ \texttt{(}x\ \texttt{+}\ z\texttt{))} & & \text{[One Step]} \\
\texttt{(}z\ \texttt{+}\ \texttt{(}y\ \texttt{+}\ x\texttt{))} \not\Longrightarrow \texttt{(}y\ \texttt{+}\ \texttt{(}x\ \texttt{+}\ z\texttt{))} & & \text{[Two Steps]} \\
\texttt{(}z\ \texttt{+}\ \texttt{(}y\ \texttt{+}\ x\texttt{))} \Longrightarrow^\star \texttt{(}y\ \texttt{+}\ \texttt{(}x\ \texttt{+}\ z\texttt{))} & & \text{[Two Steps]} \\
\texttt{(}z\ \texttt{+}\ \texttt{(}y\ \texttt{+}\ x\texttt{))} \not\Longrightarrow \texttt{(}z\ \texttt{+}\ \texttt{(}y\ \texttt{+}\ x\texttt{))} & & \text{[Zero Steps]} \\
\texttt{(}z\ \texttt{+}\ \texttt{(}y\ \texttt{+}\ x\texttt{))} \Longrightarrow^\star \texttt{(}z\ \texttt{+}\ \texttt{(}y\ \texttt{+}\ x\texttt{))} & & \text{[Zero Steps]}
\end{array}
$$

2.6.2 Proofs of Termination

A set of rewrite rules is **terminating** if and only if there is no infinite sequence of rewrites $e_1 \Longrightarrow e_2 \Longrightarrow e_3 \Longrightarrow e_4 \Longrightarrow \ldots$.

Proof that the set of rewrite rules for eliminating constants in well-formed formulae with constants is terminating: each one-step rewrite reduces the number of operators in the expression by one. QED

In general, termination proofs associate some non-negative integer with each expression, and show that certain rewrites reduce the integer, and all the other rewrites do not increase the integer.

Proof that no sequence of rewrites in the algebra of simple expressions contains an infinite number of commutativity rewrites: For each expression, consider the number of pairs of variable occurrences in the expression in which the lexically later variable occurs first in the expression. Call this the number of variable miss-orderings; e.g., the following are expressions and a list of their variable miss-orderings:

(*z* + (*y* + *x*))	$z >> y$, $z >> x$, $y >> x$
(*y* + (*z* + *x*))	$z >> x$, $y >> x$
(*y* + (*x* + *z*))	$y >> x$

Then each commutativity rewrite rule step reduces the number of variable miss-orderings by one, while the associativity rewrite steps do not change the number of variable miss-orderings. QED

Proof that no sequence of rewrites in the algebra of rational expressions contains an infinite number of / combination rewrites: Each / combination rewrite reduces the number of / operators in the expression, and other writes leave the number of / operators constant. Note: for certain distributive rewrites that duplicate subexpressions, it is important that the duplicated subexpressions must be polynomials that contain no / operators. QED

Proof that no sequence of rewrites in the algebra of simple expressions contains an infinite number of associativity rewrites: For each expression, consider the number of pairs of + operator occurrences such that one + operator occurs in the left operand of the other + operator. This number decreases for associativity rewrite rules and stays the same for commutativity rewrite rules. QED

Proof that the set of rewrites of the algebra of simple expressions is terminating: By what we proved above, any sequence of rewrites contains at most a finite number of commutativity rewrites, and at most a finite number of associativity rewrites, and therefore as these are the only two kinds of rewrites available, and sequence of rewrites is finite. QED

Proof that no sequence of rewrites in the algebra of rational expressions contains an infinite number of +/ distributivity rewrites: For each expression, consider the number of pairs of + and / operator occurrences such that the / operator occurs inside an operand of the + operator. This number decreases for +/ distributivity rewrite rules and does not increase for other rewrite rules. QED

Proof that no sequence of rewrites in the algebra of polynomials contains an infinite number of distributivity rewrites: With each expression e associate a size $|e| \geq 2$ by the inductive definition:

$$\begin{array}{ll} |v| = 2 & v \in \text{variable} \\ |\texttt{(}e_1 + e_2\texttt{)}| = |e_1| + |e_2| + 1 & e_1,\ e_2 \in \text{expression} \\ |\texttt{(}e_1 \texttt{*} e_2\texttt{)}| = |e_1| \cdot |e_2| & \end{array}$$

Then distributive rewrites reduce the size of the expression and other rewrites do not increase the size. QED

Proof that the set of dnf rewrites in the algebra of propositional calculus is terminating: With each expression e associate a size $|e| \geq 2$ by the inductive definition:

$$\begin{array}{ll} |v| = 2 & v \in \text{variable} \\ |\texttt{(}e_1 \vee e_2\texttt{)}| = |e_1| + |e_2| + 1 & e_1,\ e_2 \in \text{expression} \\ |\texttt{(}e_1 \wedge e_2\texttt{)}| = |e_1| \cdot |e_2| & \\ |\texttt{(}\neg e\texttt{)}| = 2^{|e|} & e \in \text{expression} \end{array}$$

All the rewrite rules reduce this size. For example, $\texttt{(}\neg\texttt{(}e_1 \wedge e_2\texttt{))} \Longrightarrow \texttt{((}\neg e_1\texttt{)}\vee\texttt{(}\neg e_2\texttt{))}$ reduces the size if $2^{|e_1|\cdot|e_2|} > 2^{|e_1|} + 2^{|e_2|} + 1$. Without loss of generality assume $|e_1| \geq |e_2| \geq 2$. Then $2^{|e_1|\cdot|e_2|} \geq 2^{2|e_1|} \geq 2^{|e_1|+2} > 2^{|e_1|} + 2^{|e_2|} + 1$. QED.

Exercise 2.6.2.1 Prove that no sequence of rewrites in the algebra of polynomials contains an infinite number of the following kinds of rewrites:

(a) associativity rewrites
(b) factor commutativity rewrites
(c) monomial commutativity rewrites
(d) unity, nullity, zero, and numeric combination rewrites

Hint: For each of these you must use previous results, e.g., that after some point in any rewrite sequence the distributivity rules are never applied.

Exercise 2.6.2.2 Prove that no sequence of rewrites in the algebra of rational expressions contains an infinite number of the following kinds of rewrites:

(a) negation rewrites
(b) `*/` distributivity rewrites
(c) factor commutativity rewrites
(d) monomial commutativity rewrites
(e) unity, nullity, zero, divide by 1, and numeric combination rewrites
(f) divide by 0 rewrites

Hint: For (c), (d), (e), and (f) you must use previous results, e.g., that after some point in any rewrite sequence the distributivity rules are never applied.

2.6.3 Proofs of Soundness

In our theory of formal algebras, a set of rewrite rules is **sound** if and only if $e_1 \Longrightarrow^\star e_2$ implies that $e_1 = e_2$ is provable. To prove soundness, it suffices to prove that for every rewrite rule $e_1 \Longrightarrow e_2$, $e_1 = e_2$ is provable.

For example, in the algebra of simple expressions we have the rewrite rule:

$$(v_2 \texttt{ + } (v_1 \texttt{ + } e)) \Longrightarrow (v_1 \texttt{ + } (v_2 \texttt{ + } e)) \quad \text{if } v_1 << v_2 \quad [\textbf{commutativity 2}]$$

where v_1, $v_2 \in$ variable; $e \in$ simple-expression

Therefore we need the equational proof:

$$\begin{array}{cll} & (v_2 \texttt{ + } (v_1 \texttt{ + } e)) & \\ = & ((v_2 \texttt{ + } v_1) \texttt{ + } e) & \text{by associativity} \\ = & ((v_1 \texttt{ + } v_2) \texttt{ + } e) & \text{by commutativity} \\ = & (v_1 \texttt{ + } (v_2 \texttt{ + } e)) & \text{by associativity} \end{array}$$

The reason it suffices to check just equations made directly from rewrite rules is that rewrites are made from rewrite rules either by rewriting subexpressions or by taking multiple rewrite steps; and equality proofs can also substitute equal subexpressions and take multiple steps, just as rewriting can.

Exercise 2.6.3.1 Prove that the following sets of rewrite rules are sound:

(a) rules for the algebra of polynomials
(b) rules for the algebra of rational expressions
(c) dnf rules for the algebra of propositional calculus
(d) cnf rules for the algebra of propositional calculus

2.6.4 Proofs of Confluence

There are two kinds of confluence.

A set of rewrite equations is **weakly confluent** if and only if whenever $e \Longrightarrow e_1$ and $e \Longrightarrow e_2$, then there is an expression e' such that $e_1 \Longrightarrow^\star e'$ and $e_2 \Longrightarrow^\star e'$.

A set of rewrite equations is **confluent** if and only if whenever $e \Longrightarrow^\star e_1$ and $e \Longrightarrow^\star e_2$, then there is an expression e' such that $e_1 \Longrightarrow^\star e'$ and $e_2 \Longrightarrow^\star e'$.[12]

Clearly every confluent set of rewrite rules is also weakly confluent.

Theorem: Every terminating weakly confluent set of rewrite rules is also confluent.

We will not prove this here: see the literature (e.g., [Mit96]). However, since all our systems are terminating, we need only prove weak confluence in order to prove confluence.

There are two ways to prove confluence. The easy way is:

Lemma: A terminating, sound, complete system of rewrite rules is confluent.

Proof: Let $e \Longrightarrow^\star e_1$ and $e \Longrightarrow^\star e_2$. Then we can find normal forms e'_1 and e'_2 such that $e_1 \Longrightarrow^\star e'_1$ and $e_2 \Longrightarrow^\star e'_2$, because the rewrite rules are terminating. Therefore $e \Longrightarrow^\star e'_1$ and $e \Longrightarrow^\star e'_2$. Since the rules are sound, $e = e'_1$ and $e = e'_2$ are provable, and therefore $e'_1 = e'_2$ is provable. Since the rules are complete and e'_1, e'_2 are normal, then $e'_1 \equiv e'_2$, that is, the two normal forms are syntactically identical, and therefore are the same expression, which we call e'. Then $e_1 \Longrightarrow^\star e'$ and $e_2 \Longrightarrow^\star e'$ as desired. QED

All of our sets of rewrite rules are terminating and sound. The rules for the algebra's of simple expressions, integer evaluation, integer evaluation modulo 4, polynomials, and rational expressions with the common factor rule are all complete, and therefore confluent.

This leaves the algebra of rational expressions without the common factor rewrite rule and the algebra of propositional calculus with or without constants. For these we must use the harder proof of confluence, which involves the concept of critical pairs.

Consider the algebra of simple expressions. If v_1, v_2, v_3, v_4 are variables with $v_1 << v_2 << v_3 << v_4$, then we can rewrite '`((`v_2 `+` v_1`) + (`v_4 `+` v_3`))`' in the following two different ways:

[12] Confluence is also called **strong confluence**.

$$
\begin{array}{rcl}
\texttt{((}v_2\ \texttt{+}\ v_1\texttt{)}\ \texttt{+}\ \texttt{(}v_4\ \texttt{+}\ v_3\texttt{))} & \Longrightarrow & \texttt{((}v_1\ \texttt{+}\ v_2\texttt{)}\ \texttt{+}\ \texttt{(}v_4\ \texttt{+}\ v_3\texttt{))} \\
 & \Longrightarrow & \texttt{((}v_1\ \texttt{+}\ v_2\texttt{)}\ \texttt{+}\ \texttt{(}v_3\ \texttt{+}\ v_4\texttt{))} \\
\texttt{((}v_2\ \texttt{+}\ v_1\texttt{)}\ \texttt{+}\ \texttt{(}v_4\ \texttt{+}\ v_3\texttt{))} & \Longrightarrow & \texttt{((}v_2\ \texttt{+}\ v_1\texttt{)}\ \texttt{+}\ \texttt{(}v_3\ \texttt{+}\ v_4\texttt{))} \\
 & \Longrightarrow & \texttt{((}v_1\ \texttt{+}\ v_2\texttt{)}\ \texttt{+}\ \texttt{(}v_3\ \texttt{+}\ v_4\texttt{))}
\end{array}
$$

The two different subexpressions that are rewritten in the first step of these two rewrites, '$\texttt{(}v_2\ \texttt{+}\ v_1\texttt{)}$' and '$\texttt{(}v_4\ \texttt{+}\ v_3\texttt{)}$', are disjoint. These two disjoint subexpressions can be rewritten in either of two orders, either the first subexpression can be rewritten and then the second subexpression can be rewritten, or the second subexpression can be rewritten and then the first subexpression can be rewritten. This is leads to two distinct rewrites, but both rewrites get to a common point.

Thus rewriting disjoint subexpressions always gives confluent results.

Now with the same variables consider the rewrites:

$$
\begin{array}{rcl}
\texttt{(}v_2\ \texttt{+}\ \texttt{(}v_1\ \texttt{+}\ \texttt{(}v_4\ \texttt{+}\ v_3\texttt{)))} & \Longrightarrow & \texttt{(}v_1\ \texttt{+}\ \texttt{(}v_2\ \texttt{+}\ \texttt{(}v_4\ \texttt{+}\ v_3\texttt{)))} \\
 & \Longrightarrow & \texttt{(}v_1\ \texttt{+}\ \texttt{(}v_2\ \texttt{+}\ \texttt{(}v_3\ \texttt{+}\ v_4\texttt{)))} \\
\texttt{(}v_2\ \texttt{+}\ \texttt{(}v_1\ \texttt{+}\ \texttt{(}v_4\ \texttt{+}\ v_3\texttt{)))} & \Longrightarrow & \texttt{(}v_2\ \texttt{+}\ \texttt{(}v_1\ \texttt{+}\ \texttt{(}v_3\ \texttt{+}\ v_4\texttt{)))} \\
 & \Longrightarrow & \texttt{(}v_1\ \texttt{+}\ \texttt{(}v_2\ \texttt{+}\ \texttt{(}v_3\ \texttt{+}\ v_4\texttt{)))}
\end{array}
$$

Here the two subexpressions being rewritten overlap, which requires that one be inside the other. These two subexpressions are '$\texttt{(}v_2\ \texttt{+}\ \texttt{(}v_1\ \texttt{+}\ \texttt{(}v_4\ \texttt{+}\ v_3\texttt{)))}$' and its subexpression '$\texttt{(}v_4\ \texttt{+}\ v_3\texttt{)}$'. But in this case the smaller rewritten subexpression is inside a meta-variable of the larger rewrite. The rewrite for the larger subexpression is:

$$\texttt{(}v_2\ \texttt{+}\ \texttt{(}v_1\ \texttt{+}\ e\texttt{))} \Longrightarrow \texttt{(}v_1\ \texttt{+}\ \texttt{(}v_2\ \texttt{+}\ e\texttt{))} \quad \text{if } v_1 << v_2 \quad [\textbf{commutativity 2}]$$

where v_1, $v_2 \in$ variable; $e \in$ simple-expression

and the smaller subexpression is inside the meta-variable subexpression represented by e (in this case it actually equals the expression assigned to that meta-variable). All our rewrite rules just copy meta-variables, and hence the subexpressions they represent, so a rewrite done inside a subexpression represented by a meta-variable can be done before or after the rewrite containing the meta-variable is done, getting to the same place in both cases.

Thus rewriting two subexpressions, one of which is contained within a subexpression represented by a meta-variable in the rewrite rule for the other expression, gives confluent results.

We are left with one last case: one of the rewritten subexpressions is inside the other, but not inside any subexpression represented by a meta-variable in the rewrite rule that rewrites the larger expression. The results of the two rewrites in this case are called a **critical pair**.

For the algebra of simple expressions, the critical pairs are:

$$\begin{array}{lll}
\texttt{(}v_3\ \texttt{+}\ \texttt{(}v_2\ \texttt{+}\ v_1\texttt{))} & \Longrightarrow & \texttt{(}v_2\ \texttt{+}\ \texttt{(}v_3\ \texttt{+}\ v_1\texttt{))}\ \texttt{or}\ \texttt{(}v_3\ \texttt{+}\ \texttt{(}v_1\ \texttt{+}\ v_2\texttt{))} \\
\texttt{((}v_2\ \texttt{+}\ v_1\texttt{)}\ \texttt{+}\ e\texttt{)} & \Longrightarrow & \texttt{((}v_1\ \texttt{+}\ v_2\texttt{)}\ \texttt{+}\ e\texttt{)}\ \texttt{or}\ \texttt{(}v_2\ \texttt{+}\ \texttt{(}v_1\ \texttt{+}\ e\texttt{))} \\
\texttt{(}v_3\ \texttt{+}\ \texttt{(}v_2\ \texttt{+}\ \texttt{(}v_1\ \texttt{+}\ e\texttt{)))} & \Longrightarrow & \quad\texttt{(}v_2\ \texttt{+}\ \texttt{(}v_3\ \texttt{+}\ \texttt{(}v_1\ \texttt{+}\ e\texttt{)))} \\
 & & \texttt{or}\ \ \texttt{(}v_3\ \texttt{+}\ \texttt{(}v_1\ \texttt{+}\ \texttt{(}v_2\ \texttt{+}\ e\texttt{)))} \\
\texttt{((}v_2\ \texttt{+}\ \texttt{(}v_1\ \texttt{+}\ e_1\texttt{))}\ \texttt{+}\ e_2\texttt{)} & \Longrightarrow & \quad\texttt{((}v_1\ \texttt{+}\ \texttt{(}v_2\ \texttt{+}\ e_1\texttt{))}\ \texttt{+}\ e_2\texttt{)} \\
 & & \texttt{or}\ \ \texttt{(}v_2\ \texttt{+}\ \texttt{((}v_1\ \texttt{+}\ e_1\texttt{)}\ \texttt{+}\ e_2\texttt{))}
\end{array}$$

where v_1, v_2, $v_3 \in$ variable; $v_1 << v_2 << v_3$; e, e_1, $e_2 \in$ simple-expression

It turns out that all we need to do to prove confluence of a set of rewrite rules is to prove that the two expressions in a critical pair can be rewritten to a common expression, for every critical pair of the rewrite rules. For the algebra of simple expressions this proof is given by:

$$
\begin{array}{lcl}
(v_3 \texttt{ + } (v_2 \texttt{ + } v_1)) & \Longrightarrow & (v_2 \texttt{ + } (v_3 \texttt{ + } v_1)) \\
 & \Longrightarrow & (v_2 \texttt{ + } (v_1 \texttt{ + } v_3)) \\
 & \Longrightarrow & (v_1 \texttt{ + } (v_2 \texttt{ + } v_3)) \\
(v_3 \texttt{ + } (v_2 \texttt{ + } v_1)) & \Longrightarrow & (v_3 \texttt{ + } (v_1 \texttt{ + } v_2)) \\
 & \Longrightarrow & (v_1 \texttt{ + } (v_3 \texttt{ + } v_2)) \\
 & \Longrightarrow & (v_1 \texttt{ + } (v_2 \texttt{ + } v_3)) \\
((v_2 \texttt{ + } v_1) \texttt{ + } e) & \Longrightarrow & ((v_1 \texttt{ + } v_2) \texttt{ + } e) \\
 & \Longrightarrow & (v_1 \texttt{ + } (v_2 \texttt{ + } e)) \\
((v_2 \texttt{ + } v_1) \texttt{ + } e) & \Longrightarrow & (v_2 \texttt{ + } (v_1 \texttt{ + } e)) \\
 & \Longrightarrow & (v_1 \texttt{ + } (v_2 \texttt{ + } e)) \\
(v_3 \texttt{ + } (v_2 \texttt{ + } (v_1 \texttt{ + } e))) & \Longrightarrow & (v_2 \texttt{ + } (v_3 \texttt{ + } (v_1 \texttt{ + } e))) \\
 & \Longrightarrow & (v_2 \texttt{ + } (v_1 \texttt{ + } (v_3 \texttt{ + } e))) \\
 & \Longrightarrow & (v_1 \texttt{ + } (v_2 \texttt{ + } (v_3 \texttt{ + } e))) \\
(v_3 \texttt{ + } (v_2 \texttt{ + } (v_1 \texttt{ + } e))) & \Longrightarrow & (v_3 \texttt{ + } (v_1 \texttt{ + } (v_2 \texttt{ + } e))) \\
 & \Longrightarrow & (v_1 \texttt{ + } (v_3 \texttt{ + } (v_2 \texttt{ + } e))) \\
 & \Longrightarrow & (v_1 \texttt{ + } (v_2 \texttt{ + } (v_3 \texttt{ + } e))) \\
((v_2 \texttt{ + } (v_1 \texttt{ + } e_1)) \texttt{ + } e_2) & \Longrightarrow & ((v_1 \texttt{ + } (v_2 \texttt{ + } e_1)) \texttt{ + } e_2) \\
 & \Longrightarrow & (v_1 \texttt{ + } ((v_2 \texttt{ + } e_1) \texttt{ + } e_2)) \\
 & \Longrightarrow & (v_1 \texttt{ + } (v_2 \texttt{ + } (e_1 \texttt{ + } e_2))) \\
((v_2 \texttt{ + } (v_1 \texttt{ + } e_1)) \texttt{ + } e_2) & \Longrightarrow & (v_2 \texttt{ + } ((v_1 \texttt{ + } e_1) \texttt{ + } e_2)) \\
 & \Longrightarrow & (v_2 \texttt{ + } (v_1 \texttt{ + } (e_1 \texttt{ + } e_2))) \\
 & \Longrightarrow & (v_1 \texttt{ + } (v_2 \texttt{ + } (e_1 \texttt{ + } e_2)))
\end{array}
$$

where v_1, v_2, $v_3 \in$ variable; $v_1 << v_2 << v_3$; e, e_1, $e_2 \in$ simple-expression

Of course we already know that the algebra of simple expressions is confluent because it is complete.

An example of proving that a critical pair is confluent from the algebra of rational expressions without the common factor rewrite rule is:

$$\begin{array}{rcl}
\texttt{((}p_1\ \texttt{/}\ p_2\texttt{)}\ \texttt{+}\ \texttt{(}p_3\ \texttt{/}\ p_4\texttt{))} & \Longrightarrow & \texttt{((((}p_1\ \texttt{/}\ p_2\texttt{)}\ \texttt{*}\ p_4\texttt{)}\ \texttt{+}\ p_3\texttt{)}\ \texttt{/}\ p_4\texttt{)} \\
& \Longrightarrow & \texttt{((((}p_1\ \texttt{*}\ p_4\texttt{)}\ \texttt{/}\ p_2\texttt{)}\ \texttt{+}\ p_3\texttt{)}\ \texttt{/}\ p_4\texttt{)} \\
& \Longrightarrow & \texttt{((((}p_1\ \texttt{*}\ p_4\texttt{)}\ \texttt{+}\ \texttt{(}p_2\ \texttt{*}\ p_3\texttt{))}\ \texttt{/}\ p_2\texttt{)}\ \texttt{/}\ p_4\texttt{)} \\
& \Longrightarrow & \texttt{(((}p_1\ \texttt{*}\ p_4\texttt{)}\ \texttt{+}\ \texttt{(}p_2\ \texttt{*}\ p_3\texttt{))}\ \texttt{/}\ \texttt{(}p_2\ \texttt{*}\ p_4\texttt{))} \\
\texttt{((}p_1\ \texttt{/}\ p_2\texttt{)}\ \texttt{+}\ \texttt{(}p_3\ \texttt{/}\ p_4\texttt{))} & \Longrightarrow & \texttt{((}p_1\ \texttt{+}\ \texttt{(}p_2\ \texttt{*}\ \texttt{(}p_3\ \texttt{/}\ p_4\texttt{)))}\ \texttt{/}\ p_2\texttt{)} \\
& \Longrightarrow & \texttt{((}p_1\ \texttt{+}\ \texttt{((}p_2\ \texttt{*}\ p_3\texttt{)}\ \texttt{/}\ p_4\texttt{))}\ \texttt{/}\ p_2\texttt{)} \\
& \Longrightarrow & \texttt{((((}p_1\ \texttt{*}\ p_4\texttt{)}\ \texttt{+}\ \texttt{(}p_2\ \texttt{*}\ p_3\texttt{))}\ \texttt{/}\ p_4\texttt{)}\ \texttt{/}\ p_2\texttt{)} \\
& \Longrightarrow & \texttt{(((}p_1\ \texttt{*}\ p_4\texttt{)}\ \texttt{+}\ \texttt{(}p_2\ \texttt{*}\ p_3\texttt{))}\ \texttt{/}\ \texttt{(}p_4\ \texttt{*}\ p_2\texttt{))}
\end{array}$$

where p_1, p_2, p_3, $p_4 \in$ polynomial

To finish this we need to prove that for any two polynomials p_3 and p_4 there is a polynomial p such that:

$$\begin{array}{c}
\texttt{(}p_3\ \texttt{*}\ p_4\texttt{)} \Longrightarrow^\star p \\
\texttt{and} \\
\texttt{(}p_4\ \texttt{*}\ p_3\texttt{)} \Longrightarrow^\star p
\end{array}$$

Since the rewrite rules applicable to polynomials are the same in the algebras of polynomials and rational expressions, we may apply what we know about the algebra of polynomials. We know that

$$\begin{array}{c}
\texttt{(}p_3\ \texttt{*}\ p_4\texttt{)} \Longrightarrow^\star p \\
\texttt{and} \\
\texttt{(}p_4\ \texttt{*}\ p_3\texttt{)} \Longrightarrow^\star p'
\end{array}$$

for some normal form polynomials p and p' because the rewrite rules for polynomials are terminating. We know that

$$\texttt{(}p_3\ \texttt{*}\ p_4\texttt{)}\ \texttt{=}\ \texttt{(}p_4\ \texttt{*}\ p_3\texttt{)}$$

is provable, and therefore since the rewrite rules for polynomials are sound, that p = p' is provable. Then since the rewrite rules for polynomials are complete, $p \equiv p'$, i.e., the p and p' are syntactically identical.

You will note that our critical pair expressions contain meta-variables, and therefore represent an infinite number of expression pairs. It is important that critical pairs use meta-variables to represent the most general cases in which one expression may be rewritten in two ways that require serious proof of confluence. However, we do not attempt to state precisely in this book how to find such 'most general cases'.

Exercise 2.6.4.1 For each of the rewrite rules below:

(A) Find all critical pairs whose larger rewritten expression is rewritten by the given rules.

(B) For each critical pair found in (A), give a proof (pair of rewrites) that the two expressions in the pair can be rewritten to a common expression.

(a) `+` associativity rewrites in the algebra of rational expressions
(b) `*` associativity rewrites in the algebra of rational expressions
(c) monomial commutativity rewrites in the algebra of rational expressions
(d) `+*` distributivity rewrites in the algebra of rational expressions
(e) `*/` distributivity rewrites in the algebra of rational expressions
(f) `/` combination rewrites in the algebra of rational expressions
(g) DeMorgan's law rewrites in the set of dnf rewrites
(h) distributivity rewrites in the set of dnf rewrites
(i) double negation rewrites in the set of dnf rewrites
(j) all rewrite rules in the set of rules for eliminating constants

2.6.5 Proofs of Completeness

In our theory of formal algebras, a set of rewrite rules is **complete** whenever two normal forms are provably equal if and only if they are syntactically identical. Since syntactically identical expressions are always provably equal, the rules are complete if and only if syntactically different normal forms are not provably equal.

How do we prove, for example, that $1 + 2x + 3x^2 = 1 + 5x + 3x^2$ is not provable in the algebra of polynomials. All we do is substitute some value, say 10, in for x, and show that the values of the polynomials on each side of the equation are different.

To keep things simple, we will consider only algebras that have a sufficient set of constants. For the algebras of polynomials and rational expressions, the numbers used in the expressions will be the constants. For the algebra of propositional calculus with constants, **f** and **t** are the constants. In general an **algebra with constants** is a formal algebra with a distinguished set of expressions that are the **constants** of the algebra.

A **model** for a formal algebra with constants is a way of assigning a value that is a constant to each expression of the formal algebra that does not contain any variables such that the value of an expression does not change when a subexpression is replaced by a different subexpression with the same value. We also require that any expression consisting of just a constant by itself be assigned that constant itself as the value of the expression.

Given a model for a formal algebra with constants, a **variable assignment** is a map from variables to constants (e.g., $x \mapsto 10$, $y \mapsto 100$). Given a model and a variable assignment, we can assign a constant value to each expression of the algebra, including expressions that contain variables (e.g., $x + xy$), by replacing each variable in the expression by its assigned constant (e.g., to get $10 + 10 \cdot 100$), and taking the value of the resulting expression without variables (e.g., 1010).

An equation $e_1 = e_2$ is said to be **valid** for a model if and only if the values the model assigns to e_1 and e_2 are the same for every variable assignment.

A model is said to be **valid** for a formal algebra if and only if all the axioms of the algebra are valid for the model.

It can be shown that given a valid model for a formal algebra, all provable equations of the algebra are valid for the model.

Therefore to prove that $e_1 = e_2$ is not provable, it suffices to find a valid model and a variable assignment that assigns different values to e_1 and e_2.

Consider the algebra of polynomials with numbers. The model we use is the obvious one that assigns numbers to expressions with no variables in the usual way. This model is valid. So to show that two polynomials are not provably equal, it suffices to find a variable assignment that gives the polynomials different values.

It can be shown that two normal form polynomials that are syntactically distinct can be given different values by an appropriate variable assignment.

It is tedious to discuss the general case, so we will discuss an example. Suppose we want to show that the syntactically distinct normal form polynomials $1 + 5x + 8x^2$ and $10 + 100x + 7x^2$ can be given different values by some variable assignment, and are therefore not provably equal.

The normal form of the difference of the polynomials is $-9-95x+x^2$. The normal form of a difference of two normal form polynomials can be shown to be not syntactically identical to zero whenever the normal form polynomials are not syntactically identical.

If we take a large enough value for x, then x^2 will be larger than the absolute value of $-9 - 95x$, so the latter cannot cancel the former, and $-9 - 95x + x^2$ will not be zero. QED

By such methods can it be shown that distinct normal form polynomials may be given different values by appropriate variable assignments, and so are not provably equal. Thus the algebra of polynomials is complete.

To show that the algebra of rational expressions with the common factor rewrite rule and only integer numbers is complete, one must show that if $\frac{p_1}{p_2} = \frac{p'_1}{p'_2}$ for all variable assignments, if p_1 and p_2 have no common factors, and if p'_1 and p'_2 have no common factors, then $p_1 = p'_1$ and $p_2 = p'_2$ for all variable assignments, and therefore $p_1 \equiv p'_1$ and $p_2 \equiv p'_2$ if all polynomials are in normal form. Without going into details, we note that the set of polynomials with integer coefficients is what is called a '**unique factorization domain**' by the mathematicians. From the hypotheses we get that $p_1 \cdot p'_2 = p'_1 \cdot p_2$. Therefore each factor of p_1 must divide $p'_1 \cdot p_2$, but since it cannot divide p_2, it must divide p'_1. Thus every factor of p_1 divides p'_1 and so p_1 divides p'_1. By similar reasoning p'_1 divides p_1, and therefore $p_1 = p'_1$. Similar reasoning gives $p_2 = p'_2$.

Exercise 2.6.5.1 Prove that the following are complete:

(a) the algebra of integer evaluation modulo 4
(b) the algebra of simple expressions

2.6.6 Viewpoint and Terminology

The subject matter of this section is discussed in detail in [Mit96], but as with the literature in general, the viewpoint and terminology are quite different from ours. The viewpoint in the literature originates from mathematics, where the important thing is the model, which is viewed as a set of values and a collection of operator functions. The formal algebra, which is a language, is viewed as less important. The mathematical models, which are much more general than our models (they can assign values that are not constants in expressions), are introduced and studied first, and are called '**algebras**'. Our formal algebra is introduced as a peculiar kind of algebra in which the value of an expression is the expression itself. This is called a '**term algebra**' (where 'term' means 'expression'), a '**free algebra**', or a '**universal algebra**'.

But in computer science it is the set of expressions viewed as a language that is the important thing, and therefore we introduce it first. Models are not important to us here, except for proving completeness.

The literature also has many uses for the terms ‘sound’ and ‘complete’, but does not discuss our concepts of sound and complete sets of rewrite rules or give these concepts names. In some generality an algorithm that gives yes or no answers or that might loop indefinitely is said to be ‘**sound**’ if it does not give any yes answer which should really be no, and is said to be ‘**complete**’ if it always gives a yes answer when it really should (and thus never gives an incorrect no answer or loops indefinitely when it should give a yes). With this in mind, soundness of rewrite rules as we have defined it is the obvious way to prove soundness of the equation proof algorithm, and completeness of rewrite rules as we have defined it is necessary and sufficient to prove completeness of the equation proof algorithm.

Lastly, in the algebra of rational expressions we introduced equations with side conditions, such as

$$\texttt{(}x\texttt{ / (}y\texttt{ / }z\texttt{)) = ((}x\texttt{ * }z\texttt{) / }y\texttt{)} \quad \text{if not } z \texttt{ = 0} \quad \text{[/ combination]}$$

We have not attempted to deal precisely with such equations, and we do not know of any textbook that deals with such equations.

Chapter 3

Search

One of the characteristics of the intelligent mind is the ability to **search** for solutions. We wish to look at this search activity from the computer science point of view.

To the computer scientist, the mind is a memory and a choice mechanism. The memory holds information about the world. The value of the memory at any given instant is called the '**state**' of the mind. The choice mechanism looks at the current state of the mind, and decides which of several '**transitions**' to make. Each transition changes the state of the mind. Some transitions may be associated with actions that affect the outside world, and these actions are called the '**outputs**' of the transition. Some transitions may be associated with actions that obtain information from the outside world, and these actions are called the '**inputs**' of the transition. The original state of the mind when the transition was chosen is called the '**source**' of the transition, and the state of the mind after the transition is made is called the '**destination**' of the transition. The destination depends upon the source, the transition, and any inputs of the transition.

In the context of a search, transitions may be called '**choices**' or '**moves**'.

The goal of the search is to reach some state that represents a solution to the problem to be solved. These solution states are simply called '**goals**'.

In real problems, the state of the mind that is important may just be what the mind knows about the state of the outside world, and so the state of the mind may just be part of the state of the world.

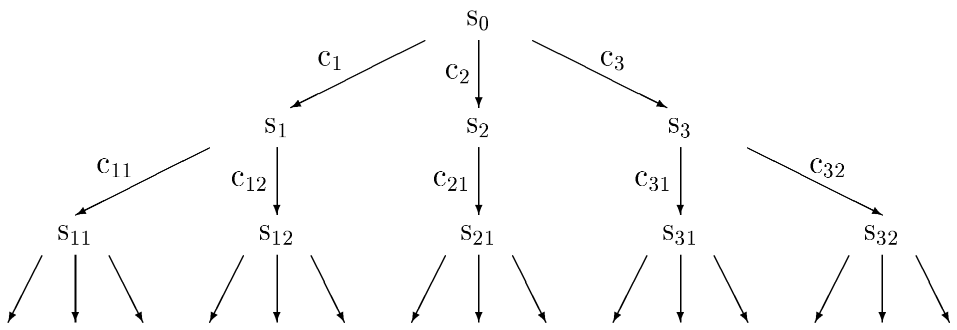
In this chapter we first investigate two kinds of search: goal search and game search. We then introduce another kind of theorem proving, resolution theorem proving, that amounts to a goal search in a hypergraph.

3.1 Goal Search

3.1.1 Goal Search Trees

The search activity described above can be represented by a labeled tree. The nodes of the tree are labeled by states of the mind. The arrows of the tree correspond to choices. At a given node, each choice possible for the state labeling the node corresponds to an arrow to a child of the node, and the label of the child is the state reached by making that choice. The root of the tree (the top node) is labeled with the state at the beginning of the search.

Pictorially the tree is:



Here the $s_{\dots}$ are states that label nodes, and the $c_{\dots}$ are choices that label arrows. The root is labeled by the initial state s_0.

Many nodes of the tree may be labeled by the same state.

A node labeled with a goal state is a leaf (has no children) because once attained the search stops. Other nodes may be leaves that are dead ends, from which no choices are possible, even though the node's states are not goals.

The nodes reachable from a given node by following a single arrow are called the **children** of the given node. A node is the **parent** of its children.

The nodes reachable from a given node by following a path of zero or more arrows are called the **descendents** of the given node. If a first node is a descendent of a second node, then the second node is an **ancestor** of the first node.

Every node but the root node has exactly one parent, but the root node has zero parents. Every node is a descendent of the root node, but the root node is a descendent only of itself.

Recall that a tree has no **cycles**; that is, you cannot start at a node and by following arrows return to the same node.

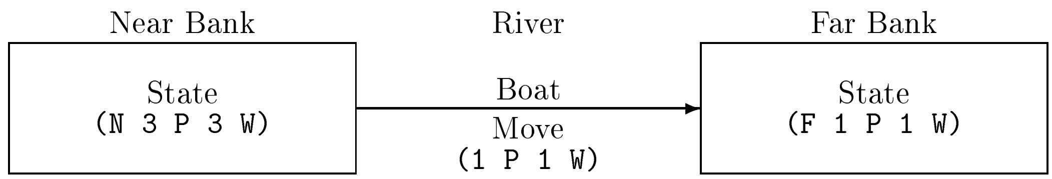
3.1.2 The Picnic Problem

In the **picnic problem** 3 pigs and 3 wolves going on a picnic must cross a river using a boat. The pigs must never be outnumbered by the wolves, either on one side of the river, or in the boat, else the wolves will be overcome by temptation and eat the pigs. The boat holds at most 2 animals, but requires at least one animal to row.

The states of this search correspond to the states of the world. A picnic problem state is defined by knowing which side of the river the boat is on, the number of pigs on that side, and the number of wolves on that side. The two sides can be named 'near' and 'far'. We use a LISP s-expression to encode that state: for example, `(N 3 P 2 W)` says the boat, 3 pigs, and 2 wolves are on the near (`N`) side of the river. Given this information, we can deduce that 0 pigs and 1 wolf are on the far side of the river.

A transition corresponds to sending the boat across the river with some animals in it, and can be labeled by the number of pigs and wolves in the boat. For example, `(1 P 0 W)` means that 1 pig and 0 wolves are in the boat.

An example transition is:



The set of states and transitions (which in this case are also called '**moves**') can be defined by the grammar:

state	::=	(boat's-side number-pigs-on-boat's-side `P` number-wolves-on-boat's-side `W )`
boat's-side	::=	`N` | `F` [Near | Far]
number-of-pigs-on-boat's-side	::=	`0` | `1` | `2` | `3`
number-of-wolves-on-boat's-side	::=	`0` | `1` | `2` | `3`
move	::=	(number-pigs-in-boat `P` number-wolves-in-boat `W )`
number-of-pigs-in-boat	::=	`0` | `1` | `2`
number-of-wolves-in-boat	::=	`0` | `1` | `2`

Not all the states and moves defined by this grammar are legal. For example, `(N 2 P 3 W)` is illegal because it has more wolves than pigs. `(F 1 P 0 W)` is illegal for the more subtle reason that on the near bank there must be 2 pigs and 3 wolves. The move `(2 P 2 W)` is illegal because there are two many animals in the boat.

A move from a legal source state can also be illegal because it leads to an illegal destination state. Some examples of legal and illegal moves are:

Source State	Move	Destination State	Is Destination Legal?
`( N 3 P 3 W )`	`( 1 P 1 W )`	`( F 1 P 1 W )`	Yes
`( N 3 P 3 W )`	`( 1 P 0 W )`	`( F 1 P 0 W )`	No
`( F 1 P 1 W )`	`( 1 P 0 W )`	`( N 3 P 2 W )`	Yes
`( F 1 P 1 W )`	`( 0 P 1 W )`	`( N 2 P 3 W )`	No

In general we need to define the set of legal moves from any given legal state. These legal moves will then be the transitions from the state, and their destinations will be the children of the state in the search tree. The definition of a legal move, assuming the source state is legal, is given by the following:

$$\boxed{\begin{array}{c}\text{State}\\ (s_1\ p_1\ \mathtt{P}\ w_1\ \mathtt{W})\end{array}} \xrightarrow[(p_m\ \mathtt{P}\ w_m\ \mathtt{W})]{\text{Move}} \boxed{\begin{array}{c}\text{State}\\ (s_2\ p_2\ \mathtt{P}\ w_2\ \mathtt{W})\end{array}}$$

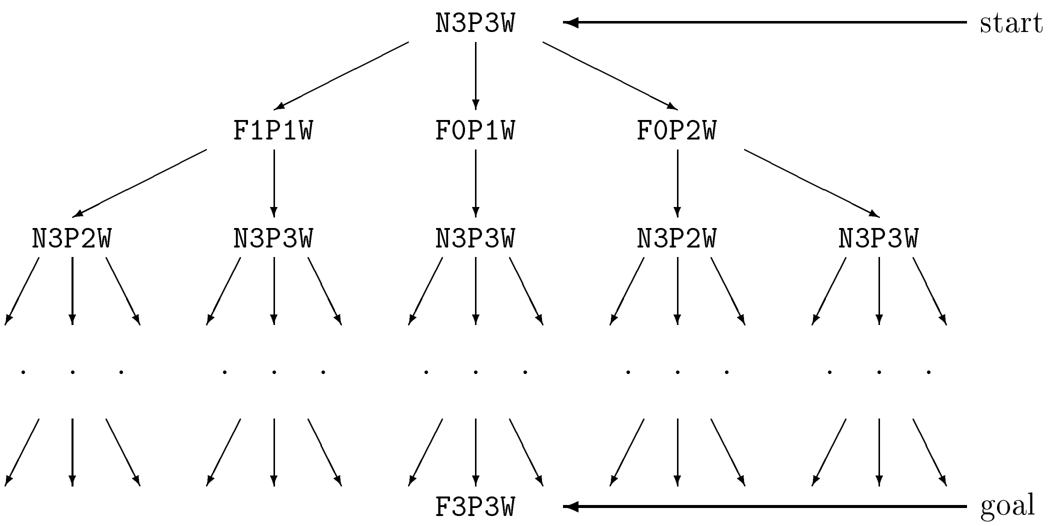
where:

$$\begin{array}{lll}
s_2 = \mathtt{F} & \text{if } s_1 = \mathtt{N} & [\text{Flip } \mathtt{N}/\mathtt{F}] \\
 = \mathtt{N} & \text{if } s_1 = \mathtt{F} & \\
p_2 = 3 - p_1 + p_m & & \\
w_2 = 3 - w_1 + w_m & & \\
0 \le p_2 \le 3,\ 0 \le w_2 \le 3 & & \\
 & & \\
w_2 \le p_2 \text{ or } p_2 = 0 & & [\text{Boat's Side}] \\
 & & \\
w_1 - w_m \le p_1 - p_m \text{ or } p_1 - p_m = 0 & & [\text{Other Side}] \\
 & & \\
1 \le p_m + w_m \le 2, \quad 0 \le p_m, w_m & & [\text{Inside Boat}]
\end{array}$$

Some examples of moves that are illegal according to this definition are:

Source State	Move	Destination State	Illegal Because
N3P3W	2P0W	F2P0W	$w_1 - w_m = 3 \not\le p_1 - p_m = 1 \neq 0$
N2P1W	0P2W	F1P4W	$w_2 = 4 \not\le 3$
F1P1W	0P1W	N2P3W	$w_2 = 3 \not\le p_2 = 2 \neq 0$

where we have made the labels more compact by omitting spaces and parentheses. The search tree for the picnic problem is:



For the picnic problem we will typically make the state labels more compact by omitting spaces and parentheses. Also, we will often omit move labels, since the label of a move can be computed from the labels of the source and destination of the move.

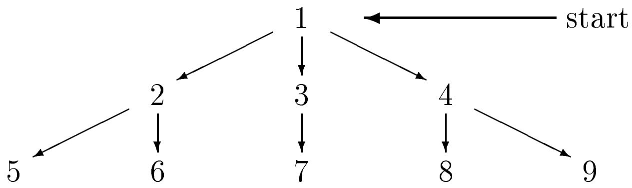
Note that a state such as `N3P3W` labels many different nodes of this search tree. For any move from a state, there is a corresponding backwards move, so a state appears at every other level in the tree downward from where the state first appears in the tree.

3.1.3 Goal Search Algorithms

The computer can look at only one node at a time in a search tree, but the computer can keep a list of the labels of nodes to be looked at, or 'visited', later. When the computer visits a node, it can check whether the node is a goal node, and it can find out the children of the node and add their labels to the list of labels for nodes to be visited later.

The strategy of some search algorithms is to visit all the nodes. The **breadth first** search algorithm visits each node on a given level of the tree before visiting any nodes at lower levels.

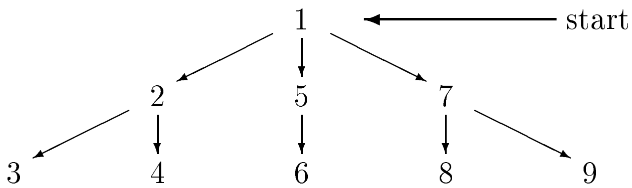
The following is a small tree in which each node is labeled by the order in which the node is visited by a breadth first search of the tree:



A simple breadth first search of the picnic problem search tree visits 10,419 nodes before finding the goal state. However, many of these nodes represent the same state, as there are fewer than $2 \cdot 4 \cdot 4 = 32$ distinct states in the tree. We can optimize the search by skipping any node whose state has already been seen: that is, we keep a list of the states that labeled the nodes we have seen and skip a node if its label is on the list. If we perform this optimization, the breadth first search succeeds after visiting only 15 nodes.

The **depth first** search algorithm visits all the children of a given node before visiting any other nodes on the same level as the given node.

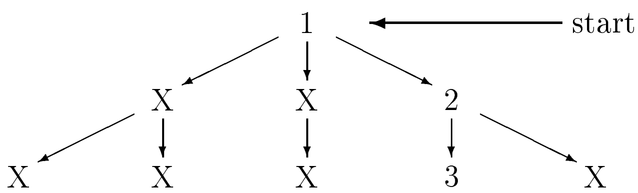
The following is a small tree in which each node is labeled by the order in which the node is visited by a depth first search of the tree:



Without the optimization just mentioned of skipping nodes whose labels are states already seen, the depth first search for the picnic problem can easily run forever without finding any goal. This is because the picnic problem search tree is actually infinite, and has an infinite number of levels. With the optimization of skipping seen states, the depth first search often runs faster than breadth first search, and for the picnic problem, depth first search succeeds after visiting only 12 nodes.

The **random** search algorithm visits a randomly chosen child of the node just visited. Thus starting at the root of the tree, a child of the root is randomly chosen for a visit, then a grandchild of the root is randomly chosen, then a great grandchild of the root is randomly chosen, and so forth.

The following is a small tree in which each node is labeled by the order in which the node might be visited by a particular random search of the tree: The nodes labeled 'X' are not visited at all by this particular random search.



The random search algorithm works surprisingly well for the picnic problem because every state is reachable from every other state by making moves, and if one makes a wrong move, one can get back to the starting state of the move. Of course, if one tries the optimization of skipping already seen nodes, this random search algorithm fails miserably, because the algorithm can no longer get back to states it has already visited. For the picnic problem random search typically succeeds after visiting on the order of 100 nodes, but the exact number of nodes visited varies widely according to the random choices made.

The breadth first search algorithm can be coded into a LISP function very simply. A function named '`breadth-search`' that takes a list of the states that label the nodes to be visited can be defined by the rewrite rules:

```
(breadth-search '()) ⟹ NIL

(breadth-search '(s . y))
        ⟹ 's                                    if (goal 's)

        ⟹ (breadth-search                       otherwise
              (append 'y (children 's)))
```
where $s \in$ state; $y \in$ list-of-states

Here '`children`' is a function that given the state labeling a node returns a list of the states labeling the children of the node. Thus in the picnic problem (see the tree on page 186): :

```
(children '(N 3 P 3 W)) ⟹ '((F 1 P 1 W) (F 0 P 1 W) (F 0 P 2 W))
(children '(F 1 P 1 W)) ⟹ '((N 3 P 2 W) (N 3 P 3 W))
(children '(F 0 P 1 W)) ⟹ '((N 3 P 3 W))
(children '(F 0 P 2 W)) ⟹ '((N 3 P 2 W) (N 3 P 3 W))
```

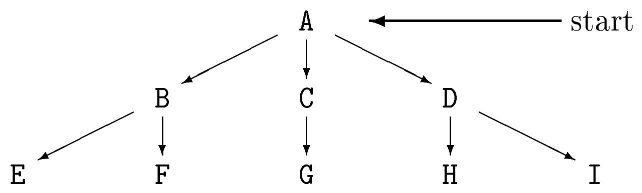
Also the '`goal`' function takes a state and returns true if and only if the state is a goal. Thus for the picnic problem:

(goal '(F 3 P 3 W)) $\Longrightarrow$ T

but '`goal`' is false for all other legal input.

'`Breadth-search`' takes the first state s off the list of labels of nodes to be visited, and adds the labels of the children of the node labeled by s to the end of the list of labels of nodes to be visited, provided the state s is not a goal. The initial list of states consists of the list whose sole element is the label of the root node of the search tree.

Thus for the tree:



with no goal node at all we have the search:

```
(breadth-search '(A)) ⟹
(breadth-search '(B C D)) ⟹
(breadth-search '(C D E F)) ⟹
(breadth-search '(D E F G)) ⟹
(breadth-search '(E F G H I)) ⟹
(breadth-search '(F G H I)) ⟹
(breadth-search '(G H I)) ⟹
(breadth-search '(H I)) ⟹
(breadth-search '(I)) ⟹
(breadth-search '()) ⟹
NIL
```

A function named '`depth-search`' that is almost identical to '`breadth-search`' can be defined by the rewrite rules:

(depth-search '()) $\Longrightarrow$ NIL

(depth-search '(s . y))
$\Longrightarrow$'s if (goal 's)

$\Longrightarrow$ (depth-search otherwise
(append (children 's) 'y))
where $s \in$ state; $y \in$ list-of-states

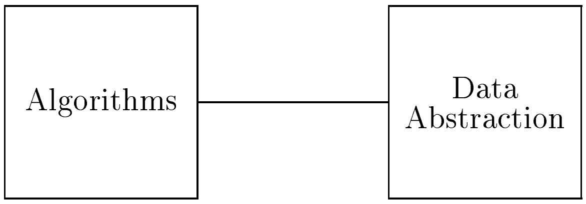
The only difference is the order of the arguments to '`append`'. For breadth first search, the children are added to the end of the list, and visited after the siblings of the children's parent node. For depth first search, the children are added to the beginning of the list, and visited before these siblings.

Both the `breadth-search` and `depth-search` functions are oversimplified in that they return only the label of the goal node when they succeed. Their caller will probably want to know the path from the root to the goal state. So real breadth first and depth first search functions will have to do some more complex bookkeeping so that they can return this path.

Exercise 3.1.3.1 Show how `(depth-search '(A))` rewrites for the search tree on page 189, after the manner in which the rewrite of `(breadth-search '(A))` was shown for that tree.

3.1.4 Goal Search Data Abstraction

Computer programs are often divided into two parts: the code that implements the algorithms, and the code that implements a '**data abstraction**' upon which the algorithms operate.



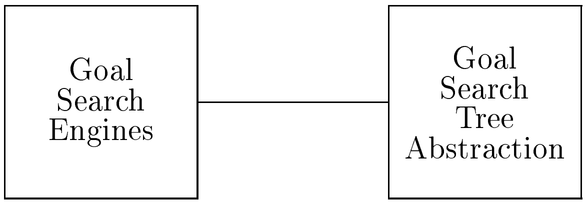
In the case of goal search algorithms, and the data abstraction is a **goal search tree**, and the code is called a '**goal search engine**'.

- state ::= s-expression
- initial-state ::= state
- '`goal`' and '`children`' are functions.
- `(goal '`s`)` $\Longrightarrow$ true if state s is a goal, false otherwise.
- `(children '`s` '`*seen*`)` $\Longrightarrow$ the list of states labeling the (unseen) children of the node labeled by state s.

  Here '*seen*' is the list of states labeling nodes already seen; these include both visited nodes and nodes on the list of nodes to be visited.

  This function may be written to use '*seen*' to prune the search tree, or it may ignore '*seen*' leaving the tree unpruned. This function may also use '*seen*' in other ways to define the tree (see resolution theorem proving).

Figure 3.1: Goal Search Tree Data Abstraction



Any set of data that can be made to look like a goal search tree can be operated on by our goal search algorithms, in principal. The problem is to make this work in in practice. This is done by an **interface**.

For a goal search tree implemented in LISP, the data abstraction is defined by the interface in Figure 3.1.

This abstraction uses the fact that almost any kind of information can be encoded in an s-expression, and therefore defining a state to be an s-expression places no limits on what can be in a state.

3.1.5 LISP Goal Search Engines

Above we described three kinds of goal search algorithms and a goal search tree data abstraction for LISP. Given this, the code for each search algorithm, the '**search engine**' for the algorithm, and is expressed as a LISP function.

The code for breadth first, depth first, and random LISP search engines is in Figures 3.2 through 3.8. There are three differences between this code and the simple `breadth-search` function we presented at the end of Section 3.1.3.

First, a list of seen states is computed so it can be passed to the '`children`' function.

Second, the path from the root of the search tree to the goal must be computed so it can be returned as the result, instead of returning just the goal state, which is not very useful by itself. Therefore the list of states to be visited in `breadth-search` is replaced by a list of paths from the root, where each path ends with a state to be visited. If one of the states to be visited turns out to be a goal state, its path can be returned.

Because of this change, the breadth first search is split across two functions. The '`breadth-first-search`' function accepts a list of states to start at, which in our usage is always a one element list containing just the root state. This function changes this list to a list of paths ending in the states it was given, and calls the '`breadth-first-search-recurse`' function that accepts a list of paths instead of a list of states. It is also possible to call '`breadth-first-search`' with a list of more than one starting state, but then the search tree becomes a search forest: i.e. a set of trees each with its own root, and we chose not to deal with the minor extra complexity involved in thinking about this case.

The paths in this code are LISP lists of states that are in reverse order, with the root state at the end of the LISP list and the end of the path at the beginning of the LISP list. This makes it easy to grow the path at its end, which is the LISP list's beginning.

A helper function, `mappush`, is used to turn a list of children into a list of paths. This function takes as arguments the list of children and a reversed path to their parent, and produces a list of reversed paths ending in the children. Thus, for example, if `E` and `F` are children of a node `B` reachable from root `A` by the non-reversed path `(A B)`, then:

```
(mappush '(E F) '(B A)) ⟹ '((E B A) (F B A))
```

produces a list of reversed paths to the children `E` and `F` of `B`.

The third change is that the breadth first search code is instrumented and can print out each state as it processes (visits) the state. The instrumentation also counts the number of visits, and terminates the search if this exceeds the value of the global variable '`*limit*`'. Because it is difficult to include instrumentation such as this in rewrite rules, the instrumentation is included in the final code only, and not in the rewrite rules.

The depth first search engine code is identical to the breadth first code, except for the order of arguments passed to the '`append`' function, as explained at the end of Section 3.1.3. The random search engine code is a straight forward implementation of the random search algorithm.

```
;;;; Search Engines
;;;;
;;;; File:         search.lsp

;; Variables to gather statistics on searches.
;;
(defvar *trace* nil)    ; True to trace.
(defvar *visits*)       ; Number of state visits.
(defvar *limit* 100)    ; Limit on number of state

;; Given a value and a header, princ the header and prin1
;; the value, all on a fresh line.  Then return the value.
;;
(defun trace-value (header value)
  (cond (*trace*
         (fresh-line)
         (princ header) (prin1 value) (fresh-line))))

;; (search-time e) evaluates e and prints timing statistics
;; on its execution.  These include the number of state visits
;; made.  The result of evaluating e is returned.
;;
(defmacro search-time (e)
  `(let ((result nil))
    (setf result (time ,e))
    (fresh-line)
    (terpri)
    (princ '|Number of States Visited: |)
    (princ *visits*)
    (terpri)
    (terpri)
    result))

;; Function to make a list of paths from a list of
;; states and a single path, by putting each state in
;; the state list at the beginning of the single path.
;;
(defun mappush (state-list path)
  (mapcar #'(lambda (state) (cons state path))
          state-list))
```

Figure 3.2: Goal Search Engine Helper Functions

```
;; Breadth first search.
;;
;; (breadth-first-search 'start-state-list
;;                       #'children #'goal)
;;
;;      ===> <path to goal> if success
;;
;;      ===> NIL if failure
;;
;;      ===> '|Limit Reached, Giving Up|
;;              if number-of-state-visits >= *limit*
;;
(defun breadth-first-search
          (start-state-list children goal)
  (setf *visits* 0)
  (breadth-first-search-recurse
   (mappush start-state-list nil)
   start-state-list
   children goal))

;; Depth first search.
;;
;; (depth-first-search 'start-state-list
;;                     #'children #'goal)
;;      ===> <same as breadth-first-search above>
;;
(defun depth-first-search
          (start-state-list children goal)
  (setf *visits* 0)
  (depth-first-search-recurse
   (mappush start-state-list nil)
   start-state-list
   children goal))
```

Figure 3.3: `Breadth-First-` and `Depth-First-Search`

```
;; In the following, instead of using a list of
;; starting states, a list of paths to starting states
;; is used, so we can keep track of the path to each
;; state.
;;
;; RR: (breadth-first-search-recurse
;;               '() 'seen #'children #'goal)
;;           ===> nil
;;     (breadth-first-search-recurse
;;                '((state . more-states) . more-paths)
;;                'seen #'children #'goal)
;;           ===> '(state . more-states)
;;                        if (funcall #'goal state)
;;                                ===> true
;;           ===> (breadth-first-search-recurse
;;                  (append 'more-paths
;;                          (mappush
;;                            'new-states
;;                            '(state .  more-states)))
;;                  (append 'seen 'new-states))
;;                        if none of the above apply,
;;                      where
;;                      (funcall #'children 'state 'seen)
;;                          ===> 'new-states
;;
```

Figure 3.4: `Breadth-First-Search-Recurse` Rewrite Rules

```
(defun breadth-first-search-recurse
            (path-list seen children goal)
  (cond
   ((null path-list) nil)
   (t
    (let* ((current-path (first path-list))
           (current-state (first current-path)))

     (trace-value '|Processing State: | current-state)
     (setf *visits* (+ *visits* 1))

     (cond

      ((funcall goal current-state)
       current-path)

      ((>= *visits* *limit*)
       '|Limit Reached, Giving Up|)

      (t
       (let ((new-states
              (funcall children current-state seen)))

        (trace-value '|New States: | new-states)

        (breadth-first-search-recurse
         (append (rest path-list)
                 (mappush new-states current-path))
         (append seen new-states)
         children goal))))))))
```

Figure 3.5: `Breadth-First-Search-Recurse` Code

```
;; Just like breadth-first-search but new paths
;; are put in front of old paths.
;;
(defun depth-first-search-recurse
          (path-list seen children goal)
  (cond

   ((null path-list) nil)

   (t
    (let* ((current-path (first path-list))
           (current-state (first current-path)))

      (trace-value '|Processing State: | current-state)
      (setf *visits* (+ *visits* 1))

      (cond

       ((funcall goal current-state)
        current-path)

       ((>= *visits* *limit*)
        '|Limit Reached, Giving Up|)

       (t
        (let ((new-states
               (funcall children current-state seen)))

         (trace-value '|New States: | new-states)

         (depth-first-search-recurse
          (append (mappush new-states current-path)
                  (rest path-list))
          (append seen new-states)
          children goal))))))))
```

Figure 3.6: `Depth-First-Search-Recurse` Code

```
;; Random search.
;;
;; (random-search 'start-state-list #'children #'goal)
;;      ===> <same as breadth-first-search above>
;;
(defun random-search (start-state-list children goal)
  (setf *visits* 0)
  (random-search-recurse
   start-state-list nil children goal))

;; RR: (random-search-recurse
;;               '() 'path #'children #'goal)
;;          ===> nil
;;     (random-search-recurse
;;               '(state . more-states) 'path
;;               #'children #'goal)
;;          ===> '(current-state . path)
;;                    if (funcall #'goal 'current-state)
;;                                ===> true
;;          ===> (random-search-recurse
;;                  (funcall #'children 'current-state
;;                           (cons 'current-state 'path))
;;                  (cons 'current-state 'path)
;;                    #'children #'goal)
;;                    otherwise
;;          where (random-element '(state . more-states))
;;                        ===> current-state
;;
;; Return a random member of a list.  The list must not
;; be of zero length.
;;
(defun random-element (list)
  (let* ((length (length list))
         (index (random length)))
        (nth index list)))
```

Figure 3.7: Random-Search

```
(defun random-search-recurse
          (state-list path children goal)

  (cond

   ((null state-list)
    nil)

   (t
    (let ((current-state (random-element state-list)))

      (trace-value '|Processing State: | current-state)
      (setf *visits* (+ *visits* 1))

      (cond

       ((funcall goal current-state)
        (cons current-state path))

       ((>= *visits* *limit*)
        '|Limit Reached, Giving Up|)

       (t
        (let ((new-states
                (funcall children current-state
                         (cons current-state path))))

         (trace-value '|New States: | new-states)

         (random-search-recurse new-states
                                (cons current-state path)
                                children goal))))))))
```

Figure 3.8: Random-Search-Recurse Code

3.1.6 Picnic Problem Results

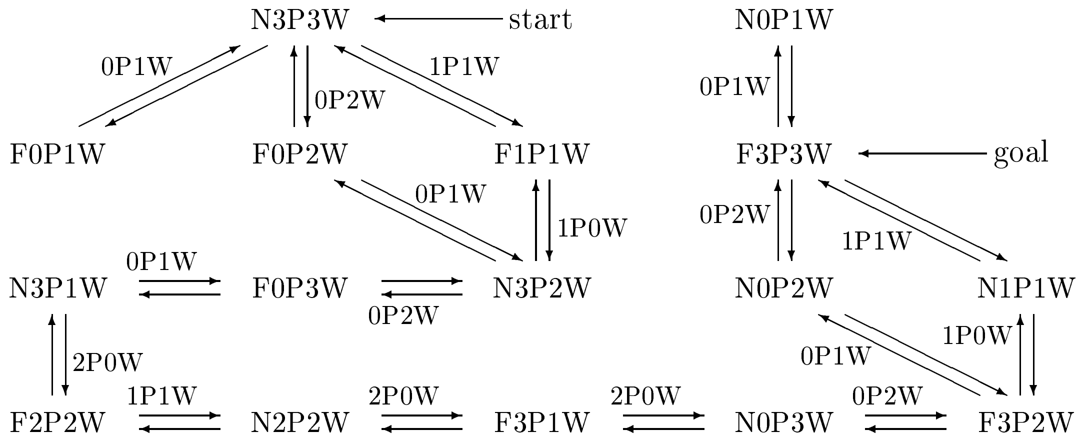
Figures 3.9 through 3.17 give an implementation of the picnic problem data abstraction and results of running the three search algorithms on the picnic problem. A summary of these results is:

- Without tree pruning, if `children` returns all children, unseen or seen, according to the rewrite rules in Figure 3.9:
  - `random-search` succeeds in 50 to 1,000 moves (Figures 3.12 and 3.13).
  - `breadth-first-search` succeeds in 10,418 moves (Figures 3.10 and 3.14).
  - `depth-first-search` gets caught in an endless loop (Figure 3.11).
- With tree pruning, if `children` returns <u>unseen</u> children only (Figure 3.15):
  - `random-search` fails quickly (gets stuck in deadend) (Figure 3.17).
  - `breadth-first-search` succeeds in 14 moves (Figure 3.16).
  - `depth-first-search` succeeds in 11 moves (Figure 3.16).

Note that depth first is more dependent on choice order than is breadth first.

3.1.7 Picnic Problem Graphs

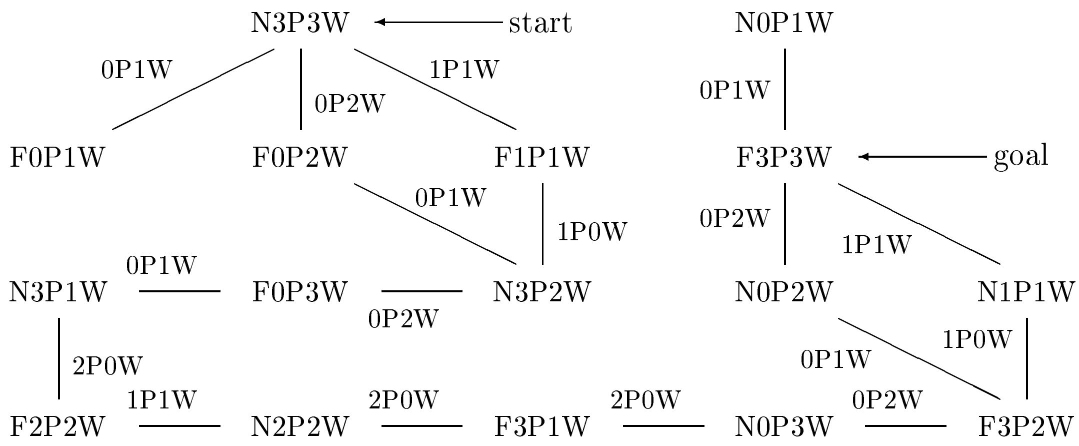
If one takes the picnic problem goal search tree and merges all nodes with the same label one gets the directed graph:



The nodes of this graph are not just labeled by states: the nodes are states. The arrows represent moves between states. In this case, all arrows occur in reverse pairs with the same label on each arrow of the pair, so in the picture of the graph one arrow label is given for each pair of arrows. A path through the search tree corresponds to a path through this graph that follows the arrows.

It is easy to see from this graph that the shortest path from start to goal has 11 moves. Breadth first search, which finds shortest paths, therefore succeeds in finding a goal on the twelfth level of the unoptimized search tree. Since we know from Figure 3.14 that the search succeeds in 10,419 visits, this gives us the approximate size of the first 12 levels of the search tree. If every node has the same number of children, b, and if the 10,419 nodes included all the nodes in the 12 levels, then we would have $10,419 = \sum_{i=0}^{11} b^i$, which can be solved to get $b = 2.194$. b is called the **branching factor** of the tree.

It seems wasteful to draw all arrows in pairs, so if we redraw the above directed graph replacing every pair of arrows by a single line segment, we get:



This is an undirected graph, and moves can be made in either direction along a line segment (or edge) connecting two nodes.

The relationship between the picnic search tree and these graphs is that each node of the tree corresponds to a unique path through the graph from the start node. A goal search can be viewed either as a search in a tree or a search in the corresponding graph. Of course, if each state appears at most once in the search tree, the graph will equal the tree.

```
;;;; The Picnic Problem
;;;;
;;;; File:          picnic.in
;;;; Author:        CS 51 (Bob Walton)
;;;; Version:       1

(load "search.lsp")
--->
;; Loading file search.lsp ...
;; Loading of file search.lsp is finished.
T

(defun goal (state) (equal state `(F 3 P 3 W)))
--->
GOAL

;; RR: (children 'state 'seen) ===>
;;         (children 'state 'seen
;;                   '((1 P 0 W) (0 P 1 W) (2 P 0 W )
;;                                (0 P 2 W) (1 P 1 W))
;;                   nil)
;;     (children 'state 'seen '() 'result) ===> 'result
;;     (children 'state 'seen '( move . more-moves )
;;                             'result)
;;         ===> (children 'state 'seen
;;                        'more-moves (cons 's 'result))
;;                if move is legal and leads to the
;;                   state s
;;         ===> (children 'state 'seen 'more-moves
;;                                     'result)
;;                if move is NOT legal
;;
;; This version ignores the SEEN argument.
;;
```

Figure 3.9: Picnic Problem `Children` Rewrite Rules

```
(defun children
   . . code omitted: like version below
                     but does not use seen )

;; Beginning of tests.

(setf *limit* 25)
--->
25
(setf *trace* t)
--->
T

(search-time (breadth-first-search
               '((N 3 P 3 W)) #'children #'goal))
--->
Processing State: (N 3 P 3 W)
New States: ((F 1 P 1 W) (F 0 P 2 W) (F 0 P 1 W))
Processing State: (F 1 P 1 W)
New States: ((N 3 P 3 W) (N 3 P 2 W))
Processing State: (F 0 P 2 W)
New States: ((N 3 P 3 W) (N 3 P 2 W))
Processing State: (F 0 P 1 W)
New States: ((N 3 P 3 W))
Processing State: (N 3 P 3 W)
New States: ((F 1 P 1 W) (F 0 P 2 W) (F 0 P 1 W))
Processing State: (N 3 P 2 W)
New States: ((F 0 P 3 W) (F 0 P 2 W) (F 1 P 1 W))
Processing State: (N 3 P 3 W)
New States: ((F 1 P 1 W) (F 0 P 2 W) (F 0 P 1 W))
Processing State: (N 3 P 2 W)
New States: ((F 0 P 3 W) (F 0 P 2 W) (F 1 P 1 W))
Processing State: (N 3 P 3 W)
New States: ((F 1 P 1 W) (F 0 P 2 W) (F 0 P 1 W))

. . . output omitted . . .

Real time: 0.016619f0 sec.
Run time: 0.016592f0 sec.
Space: 105368 Bytes

Number of States Visited: 25

|Limit Reached, Giving Up|
```

Figure 3.10: Breadth First Search Trace

```
(search-time
  (depth-first-search '((N 3 P 3 W)) #'children #'goal))
--->
Processing State: (N 3 P 3 W)
New States: ((F 1 P 1 W) (F 0 P 2 W) (F 0 P 1 W))
Processing State: (F 1 P 1 W)
New States: ((N 3 P 3 W) (N 3 P 2 W))
Processing State: (N 3 P 3 W)
New States: ((F 1 P 1 W) (F 0 P 2 W) (F 0 P 1 W))
Processing State: (F 1 P 1 W)
New States: ((N 3 P 3 W) (N 3 P 2 W))
Processing State: (N 3 P 3 W)
New States: ((F 1 P 1 W) (F 0 P 2 W) (F 0 P 1 W))
Processing State: (F 1 P 1 W)
New States: ((N 3 P 3 W) (N 3 P 2 W))
Processing State: (N 3 P 3 W)
New States: ((F 1 P 1 W) (F 0 P 2 W) (F 0 P 1 W))
Processing State: (F 1 P 1 W)
New States: ((N 3 P 3 W) (N 3 P 2 W))
Processing State: (N 3 P 3 W)
New States: ((F 1 P 1 W) (F 0 P 2 W) (F 0 P 1 W))

. . . output omitted . . .

Real time: 0.015641f0 sec.
Run time: 0.015616f0 sec.
Space: 109656 Bytes

Number of States Visited: 25

|Limit Reached, Giving Up|
```

Figure 3.11: Depth First Search Trace

```
(search-time (random-search '((N 3 P 3 W)) #'children #'goal))
--->
Processing State: (N 3 P 3 W)
New States: ((F 1 P 1 W) (F 0 P 2 W) (F 0 P 1 W))
Processing State: (F 0 P 1 W)
New States: ((N 3 P 3 W))
Processing State: (N 3 P 3 W)
New States: ((F 1 P 1 W) (F 0 P 2 W) (F 0 P 1 W))
Processing State: (F 0 P 1 W)
New States: ((N 3 P 3 W))
Processing State: (N 3 P 3 W)
New States: ((F 1 P 1 W) (F 0 P 2 W) (F 0 P 1 W))
Processing State: (F 1 P 1 W)
New States: ((N 3 P 3 W) (N 3 P 2 W))
Processing State: (N 3 P 2 W)
New States: ((F 0 P 3 W) (F 0 P 2 W) (F 1 P 1 W))
Processing State: (F 1 P 1 W)
New States: ((N 3 P 3 W) (N 3 P 2 W))
Processing State: (N 3 P 3 W)
New States: ((F 1 P 1 W) (F 0 P 2 W) (F 0 P 1 W))
Processing State: (F 0 P 1 W)
New States: ((N 3 P 3 W))
Processing State: (N 3 P 3 W)
New States: ((F 1 P 1 W) (F 0 P 2 W) (F 0 P 1 W))

. . . output omitted . . .

Real time: 0.02634f0 sec.
Run time: 0.026352f0 sec.
Space: 83576 Bytes
GC: 1, GC time: 0.011712f0 sec.

Number of States Visited: 25

|Limit Reached, Giving Up|

(setf *limit* 1000000)
--->
1000000

(setf *trace* nil)
--->
NIL

. . . code to compile functions omitted . . .
```

Figure 3.12: Random Search Trace

```
(search-time
  (random-search '((N 3 P 3 W)) #'children #'goal))
--->

Real time: 0.009755f0 sec.
Run time: 0.00976f0 sec.
Space: 51536 Bytes

Number of States Visited: 134

((F 3 P 3 W) (N 0 P 2 W) (F 3 P 2 W) (N 0 P 3 W)
  (F 3 P 2 W) (N 0 P 3 W) (F 3 P 1 W) (N 2 P 2 W)
  (F 2 P 2 W) (N 2 P 2 W) (F 2 P 2 W) (N 2 P 2 W)
  (F 2 P 2 W) (N 2 P 2 W) (F 2 P 2 W) (N 2 P 2 W)
  (F 3 P 1 W) (N 2 P 2 W) (F 2 P 2 W) (N 2 P 2 W)
  (F 2 P 2 W) (N 3 P 1 W) (F 2 P 2 W) (N 3 P 1 W)
  (F 2 P 2 W) (N 3 P 1 W) (F 2 P 2 W) (N 2 P 2 W)
  (F 2 P 2 W) (N 3 P 1 W) (F 2 P 2 W) (N 2 P 2 W)
  (F 3 P 1 W) (N 0 P 3 W) (F 3 P 1 W) (N 0 P 3 W)
  (F 3 P 1 W) (N 2 P 2 W) (F 2 P 2 W) (N 2 P 2 W)
  (F 2 P 2 W) (N 3 P 1 W) (F 2 P 2 W) (N 3 P 1 W)
  (F 2 P 2 W) (N 2 P 2 W) (F 2 P 2 W) (N 2 P 2 W)
  (F 3 P 1 W) (N 2 P 2 W) (F 2 P 2 W) (N 3 P 1 W)
  (F 0 P 3 W) (N 3 P 2 W) (F 0 P 3 W) (N 3 P 2 W)
  (F 0 P 2 W) (N 3 P 2 W) (F 0 P 3 W) (N 3 P 1 W)
  (F 2 P 2 W) (N 3 P 1 W) (F 2 P 2 W) (N 2 P 2 W)
  (F 2 P 2 W) (N 3 P 1 W) (F 2 P 2 W) (N 3 P 1 W)
  (F 0 P 3 W) (N 3 P 2 W) (F 1 P 1 W) (N 3 P 3 W)
  (F 0 P 1 W) (N 3 P 3 W) (F 0 P 1 W) (N 3 P 3 W)
  (F 0 P 1 W) (N 3 P 3 W) (F 0 P 1 W) (N 3 P 3 W)
  (F 0 P 2 W) (N 3 P 2 W) (F 0 P 3 W) (N 3 P 1 W)
  (F 0 P 3 W) (N 3 P 1 W) (F 2 P 2 W) (N 3 P 1 W)
  (F 0 P 3 W) (N 3 P 1 W) (F 0 P 3 W) (N 3 P 2 W)
  (F 1 P 1 W) (N 3 P 2 W) (F 0 P 2 W) (N 3 P 3 W)
  (F 0 P 2 W) (N 3 P 3 W) (F 0 P 2 W) (N 3 P 2 W)
  (F 1 P 1 W) (N 3 P 2 W) (F 1 P 1 W) (N 3 P 2 W)
  (F 0 P 3 W) (N 3 P 1 W) (F 0 P 3 W) (N 3 P 2 W)
  (F 0 P 2 W) (N 3 P 3 W) (F 1 P 1 W) (N 3 P 2 W)
  (F 0 P 3 W) (N 3 P 2 W) (F 0 P 3 W) (N 3 P 2 W)
  (F 1 P 1 W) (N 3 P 2 W) (F 1 P 1 W) (N 3 P 3 W)
  (F 0 P 1 W) (N 3 P 3 W) (F 1 P 1 W) (N 3 P 2 W)
  (F 1 P 1 W) (N 3 P 3 W) (F 1 P 1 W) (N 3 P 2 W)
  (F 1 P 1 W) (N 3 P 3 W) (F 0 P 2 W) (N 3 P 2 W)
  (F 0 P 2 W) (N 3 P 3 W)
)
```

Figure 3.13: Random Search Results

```
(search-time
  (breadth-first-search
   '((N 3 P 3 W)) #'children #'goal))
--->

Real time: 382.93362f0 sec.
Run time: 294.35086f0 sec.
Space: 3244041200 Bytes
GC: 4221, GC time: 218.4493f0 sec.

Number of States Visited: 10419

((F 3 P 3 W) (N 0 P 2 W) (F 3 P 2 W) (N 0 P 3 W)
  (F 3 P 1 W) (N 2 P 2 W) (F 2 P 2 W) (N 3 P 1 W)
  (F 0 P 3 W) (N 3 P 2 W) (F 1 P 1 W) (N 3 P 3 W)
)

;; Redefine children to be the same as above but with
;; already seen check added, so a state is not returned
;; as a child if it is already on the list of seen
;; states.
;;
(defun children
  (state seen
         &optional
         (move-list '((1 P 0 W) (0 P 1 W) (2 P 0 W )
                      (0 P 2 W) (1 P 1 W)))
         (children nil))
```

Figure 3.14: Breadth First Search Results

```
  (cond
   ((null move-list)
    children)
   (t
    (let*
     ((pigs (cadr state))
      (wolves (cadddr state))
      (side (car state))
      (move (car move-list))
      (delta-pigs (car move))
      (delta-wolves (caddr move))
      (new-pigs (- pigs delta-pigs))
      (new-wolves (- wolves delta-wolves))
      (new-other-side-pigs (- 3 new-pigs))
      (new-other-side-wolves (- 3 new-wolves))
      (other-side (cond ((eql 'N side) 'F) (t 'N)))
      (new-state
       `( ,other-side ,new-other-side-pigs P
                      ,new-other-side-wolves W)))
     (cond
      ((or (< new-pigs 0)
           (< new-wolves 0)
           (< new-other-side-pigs 0)
           (< new-other-side-wolves 0)
           (> new-wolves new-pigs 0)
           (> new-other-side-wolves
              new-other-side-pigs 0)

           ;; ********ADDED THE FOLLOWING*********
           (member new-state seen :test #'equal))

       (children state seen (cdr move-list) children))
      (t
       (children state seen (cdr move-list)
                 (cons new-state children)))))))
--->
CHILDREN
```

Figure 3.15: Picnic Problem `Children` Code

```
(search-time
  (breadth-first-search
   '((N 3 P 3 W)) #'children #'goal))
--->

Real time: 0.008789f0 sec.
Run time: 0.008784f0 sec.
Space: 43032 Bytes

Number of States Visited: 15

((F 3 P 3 W) (N 0 P 2 W) (F 3 P 2 W) (N 0 P 3 W)
  (F 3 P 1 W) (N 2 P 2 W) (F 2 P 2 W) (N 3 P 1 W)
  (F 0 P 3 W) (N 3 P 2 W) (F 1 P 1 W) (N 3 P 3 W)
)

(search-time
  (depth-first-search
   '((N 3 P 3 W)) #'children #'goal))
--->

Real time: 0.007812f0 sec.
Run time: 0.007808f0 sec.
Space: 36528 Bytes

Number of States Visited: 12

((F 3 P 3 W) (N 0 P 2 W) (F 3 P 2 W) (N 0 P 3 W)
  (F 3 P 1 W) (N 2 P 2 W) (F 2 P 2 W) (N 3 P 1 W)
  (F 0 P 3 W) (N 3 P 2 W) (F 1 P 1 W) (N 3 P 3 W)
)
```

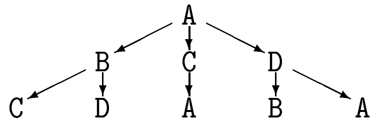
Figure 3.16: Optimized Goal Search Results

```
(setf *trace* t)
--->
T

(search-time
  (random-search '((N 3 P 3 W)) #'children #'goal))
--->
Processing State: (N 3 P 3 W)
New States: ((F 1 P 1 W) (F 0 P 2 W) (F 0 P 1 W))
Processing State: (F 1 P 1 W)
New States: ((N 3 P 2 W))
Processing State: (N 3 P 2 W)
New States: ((F 0 P 3 W) (F 0 P 2 W))
Processing State: (F 0 P 2 W)
New States: NIL

Real time: 0.001956f0 sec.
Run time: 0.001952f0 sec.
Space: 11552 Bytes

Number of States Visited: 4

NIL
```

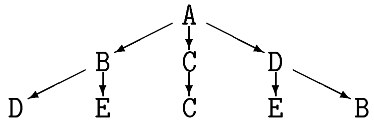
Figure 3.17: Optimized Random Goal Search Results

Exercise 3.1.7.1 For each of the following search trees, draw the corresponding directed graph with one node per state label.

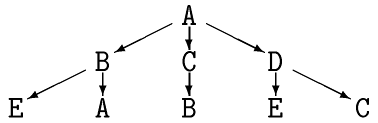
(a)



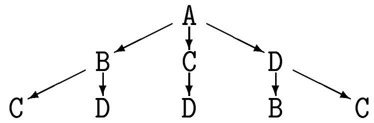
(c)



(b)



(d)



Exercise 3.1.7.2 Draw the first 3 levels of the search tree that corresponds to the picnic problem directed graph with the following graph nodes as the start node (which becomes the single root node on the first level of the tree). Do <u>not</u> prune any subtree rooted at the goal node.

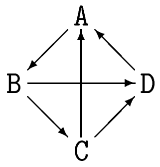
(a) `N3P2W` (b) `F1P1W` (c) `N0P3W` (d) `N0P2W`

Exercise 3.1.7.3 For each of the following directed graphs:

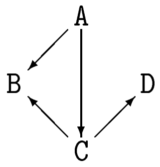
(A) Number the nodes 1, 2, 3, . . . in the order they would be visited by a breadth first search algorithm starting at node A (which gets number 1), and taking children in alphabetical order.

(B) Number the nodes 1, 2, 3, . . . in the order they would be visited by a depth first search algorithm starting at node A (which gets number 1), and taking children in alphabetical order.

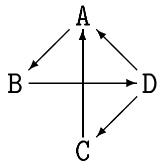
(a)



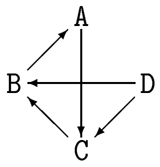
(c)



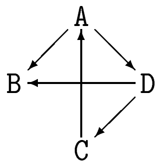
(e)



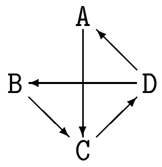
(b)



(d)



(f)

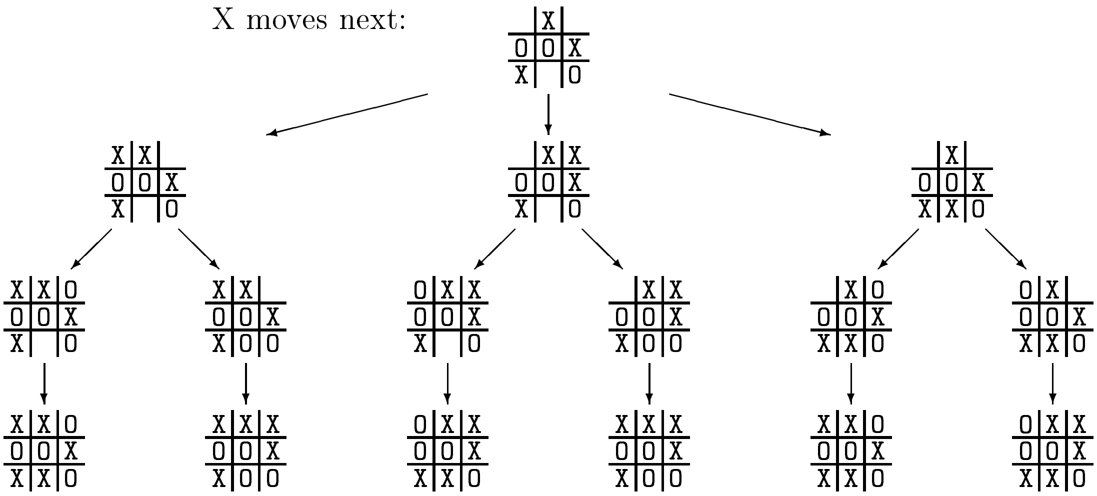


3.2 Game Search

Game search differs from goal search in that there are two minds making choices, and these minds have opposing notions of what the goal is. The two minds are called **players** or **opponents**, and an optimal choice for each player is one that will get to that player's goal no matter how the opponent makes subsequent choices. In order to pose the game search problem precisely, we will need to study game search trees in detail. We will use '**Tic Tac Toe**' as our example game.

3.2.1 Game Search Trees

The following is a **game search tree** for **Tic Tac Toe**:



The nodes in this tree are labeled by states of the game board. The two players are labeled `X` and `O`. The first and third rows of moves are to be made by `X`, the second row of moves are to be made by `O`. Some of the nodes in this tree are wins for `X`, some are wins for `O`, and some are '**draws**', which are tree leaves that are not wins for either player.

In general, in a game tree players make moves on alternating levels of the tree, some nodes are wins for some players, and leaf nodes that are not wins are draws.

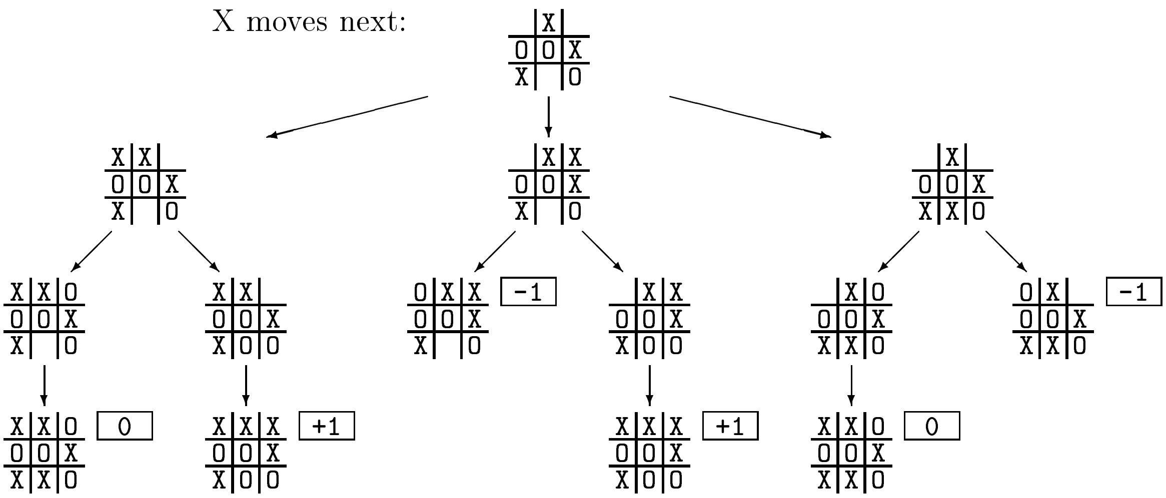
3.2.2 Assigning Values to States Directly

It is convenient to assign values to game tree nodes. Typically the values are numbers such that more positive numbers mean the node is better for one player and more negative numbers mean the node is better for the other player. For Tic Tac Toe we will assign one of three integers to each node as follows:

$$+1 \equiv \texttt{X} \text{ wins}; \qquad 0 \equiv \text{draw}; \qquad \text{-1} \equiv \texttt{O} \text{ wins}$$

We begin by assigning these values to nodes whose states are outright wins for one of the players, or nodes that are draws. We call this the '**direct**' assignment of values to game tree nodes, because the assignment depends only on the state labeling the node, and does <u>not</u> depend on the position of the node in the game tree or on assignments to children of the given node.

Direct assignment to the above game tree yields the following tree in which the value assigned to a node is displayed in a box near the node:



We have pruned the original game tree to remove children of nodes that were win's for some player. After this is done, the nodes to which a direct value assignment may be made are exactly the leaves of the tree.

3.2.3 The Minimax Algorithm

Next we consider how to assign values to the other nodes of the game tree.

Suppose we are at a node for which `X` moves next. If a child of that node has the value +1, we want it to be true that if `X` moves to that child, then `X` will definitely win, provided `X` plays smart for the rest of the game. If a child of the node has the value 0, we want it to be true that if `X` moves to that child, then `X` can draw, provided `X` plays smart for the rest of the game, but `X` cannot win if `O` plays smart for the rest of the game. And if a child of the node has the value -1, we want it to be true that if `X` moves to that child, then `X` must lose, provided `O` plays smart for the rest of the game.

If all these things are true, then an optimal move for `X` will be to one of the children with a value that is maximum among all the values of the children. And by so moving, the maximum value from that child can be assigned to the node the move started from, the parent of the child.

If we were at a node for which `O` moved next, the same argument would apply, but with the positive and negative sense of values reversed. Thus we would be looking for the minimum value of a child instead of the maximum value.

In brief, `X` tries to maximize value, and `O` tries to minimize value. Therefore we will rename the players `MAX` and `MIN`, where the player named `MAX` tries to maximize value, and the player named `MIN` tries to minimize value.

The algorithm for assigning values is thus:

- The value for a non-leaf node for which `MAX` moves next is the maximum of the values of its children.
- The value for a non-leaf node for which `MIN` moves next is the minimum of the values of its children.

This algorithm is called the '**minimax**' algorithm. Given values assigned to game tree nodes by this algorithm, it is possible to prove:[1]

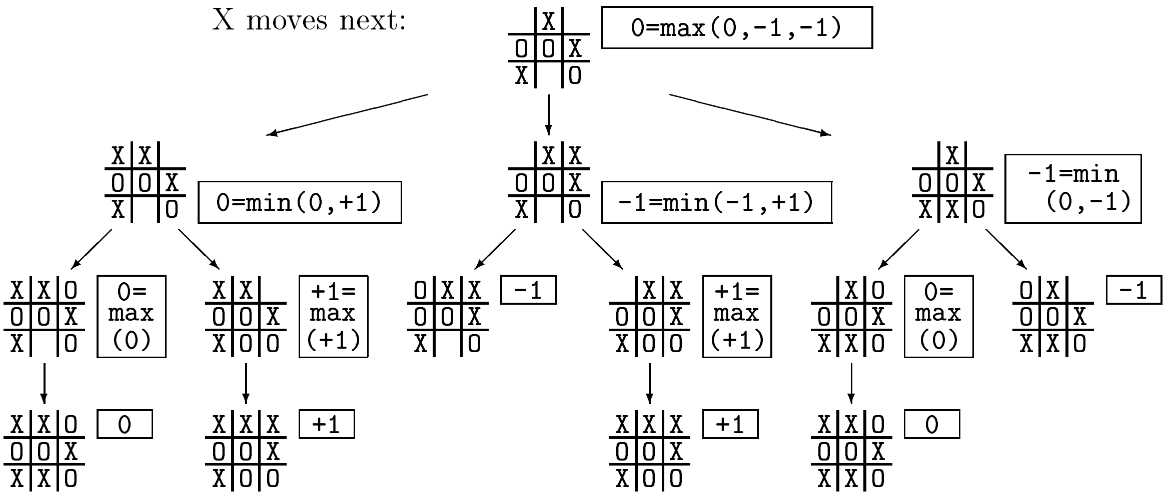
- For any game that starts at a node with value V, if `MAX` plays optimally, the game will end at a node with value equal to or greater than V.
- For any game that starts at a node with value V, if `MIN` plays optimally, the game will end at a node with value equal to or less than V.

[1]By induction on the size of the game tree.

- For a non-leaf node N whose next player is `MAX`, any move to a child with value maximum among the children of N is optimal.
- For a non-leaf node N whose next player is `MIN`, any move to a child with value minimum among the children of N is optimal.

Here 'optimal' for `MAX` means having the game end on a node with maximum possible value, and for `MIN` means having the game end on a node with minimum possible value.

Using the minimax algorithm on the game tree given above gives the following assignment of values, where the values are computed in the boxes that label each node:



In LISP, a `find-value` function can be written to find the value of a state using the minimax algorithm. The desired action of this function is:

`(find-value '`*state* `#'`*worse* `#'`*better*`)` $\Longrightarrow$ value-of-node-with-label-*state*

Here '*worse*' and '*better*' are the comparison functions that define which values are worse or better for the next player. These functions are assigned values as follows:

Next Player	`#'worse`	`#'better`
`MAX`	`#'<`	`#'>`
`MIN`	`#'>`	`#'<`

The functions '*worse*' and '*better*' define which player plays next, and can simply be switched to switch players.

Pseudocode for `(find-value '`*state* `#'`*worse* `#'`*better*`)` is:

1. If '*state*' can be given a direct value, return that value.
2. For each possible move from '*state*', compute the next state s reached by the move and call `(find-value '`*s* `#'`*better* `#'`*worse*`)` to compute a value for s.
3. Return the best of all values computed in Step 2.

Note that the '*worse*' and '*better*' arguments are switched in the recursive call in Step 2 in order to switch players.

3.2.4 Game Search Data Abstraction

Just as we defined a goal search tree by a data abstraction expressed as a LISP interface, we can define a game search tree by a data abstraction expressed as the LISP interface in Figure 3.18.

3.2.5 The Game Search Engine

Given the LISP game search tree interface, the minimax algorithm is coded in a LISP function named `find-value-and-moves`. This not only finds the value of the game search tree root node, but it finds a path of optimal moves from the root to a node whose direct value equals the value assigned to the root. The interface to this function is defined in Figure 3.19.

Code for all the functions of this interface except `find-value-and-moves` is given in Figures 3.20 through 3.22. Code for `find-value-and-moves` is left as an exercise to the reader.

3.2.6 TicTacToe Implementation

Figures 3.23 through 3.33 give an implementation of the game search tree data abstraction for **Tic Tac Toe**. This implementation is written as the file `tictactoe.in`

- state ::= s-expression
- initial-state ::= state
- move ::= s-expression
- `(state-value` *'state*`)` returns the direct value of '*state*', or `NIL` if there is no direct value.
- `(state-legal-moves` *'state*`)` returns list of legal moves from the '*state*'.
- `(state-next` *'state 'move*`)` returns the next state that results from making the '*move*' while in the '*state*'.
- `(state-next-player` *'state*`)` returns the next player to move from '*state*', which is either `'MAX` or `'MIN`
- `(state-print` *'state*`)` prints '*state*' properly formatted.
- `*state-max-value*` is the winning state value for `MAX`
- `*state-min-value*` is the winning state value for `MIN`

Figure 3.18: Game Search Tree Data Abstraction

that is processed by a LISP interpreter read-eval-print LISP loop to produce the output in the figures. Included in the file are both definitions and tests of functions. The game search engines in the file `game-search.lsp` are loaded, and tests of these search engines are placed at the end of the `tictactoe.in` file, but are not shown in the figures.

- state ::= s-expression
- move ::= s-expression   move-list ::= `NIL` | `(`move `.` move-list`)`
- value ::= s-expression
- `(find-value-and-moves '`*state* `#'`*worse* `#'`*better*`)`
  $\Longrightarrow$ `'(`*value* `.` *move-list*`)`

  `(funcall #'`*better* *v*1 *v*2`)` returns true iff `v1` is better than `v2` for the next player. `#'`*worse* is just the same as `#'`*better* but with its argument order reversed.

  '*value*' is the value of '*state*' given by the minimax algorithm, and '*move-list*' is a list of optimum moves from a node labeled by '*state*' to a node whose direct value equals '*value*'.

  Every time `find-value-and-moves` is called, it calls the function `increment-visits` below to indicate that one more node has been visited.

  Every time `find-value-and-moves` returns, it calls the function `return-value-and-moves` below to permit tracing of the results.

- `(increment-visits)` increments the integer global variable `*visits*` and if the new value exceeds the value of the integer global variable `*limit*`, calls '`error`' to announce that the limit on visits has been exceeded.
- `(return-value-and-moves '`*state* `'(`*value* `.` *move-list*`))`
  $\Longrightarrow$ `'(`*value* `.` *move-list*`)`

  Prints a trace if the global variable `*trace*` is true. The trace says that '*value*' and the move at the beginning of '*move-list*' are being returned for '*state*' by a call to `find-value-and-moves`.

- `(play-game '`*state*`)` plays a game starting at '*state*', and returns an optimal move list preceded by the outcome of the game (e.g., `MAX-WINS`). It zeros `*visits*` before the game starts.

Figure 3.19: Game Search Engine Interface

```
;;;; Game Search Engines
;;;;
;;;; File:          game-search.lsp
;;;; Author:        <your name> (<your e-mail address>)
;;;; Version:       2

;; Variables to gather statistics and traces for game
;; searches.
;;
(defvar *trace* nil)       ; True to trace.
(defvar *visits*)          ; Number of state visits
                           ; while computing one move.
(defvar *limit* 100)       ; Limit on number of state
                           ; visits while computing
                           ; one move.

;; Functions to gather statistics and trace game
;; searches.

;; Increment *VISITS* and call ERROR if it goes over the
;; *LIMIT*.
;;
(defun increment-visits ()
  (setf *visits* (+ *visits* 1))

  (cond
   ((> *visits* *limit*)
    (error "exceeded limit on state visits per move"))))
```

Figure 3.20: Game Search Global Data and Helper Functions

```
;; This function takes the STATE argument and the
;; RESULT of FIND-VALUE-AND-MOVES below and outputs
;; a trace if *TRACE* is true.  The RESULT is
;; returned, so this function can be used to return
;; the final result from FIND-VALUE-AND-MOVES.
;;
(defun return-value-and-moves
          (state value-and-moves)
  (cond
   (*trace*
    (fresh-line)
    (cond
     ((null (cdr value-and-moves))
      (princ '|Direct value |)
      (princ (car value-and-moves)))
     (t
      (princ '|Value |)
      (princ (car value-and-moves))
      (princ '| on move |)
      (princ (cadr value-and-moves))))
    (princ '| at state: |)
    (state-print state)
    (fresh-line)))
  value-and-moves)
```

Figure 3.21: `Return-Value-and-Moves`

```
;; Play a game starting at a given state.
;;
;; Returns a list of all the moves made preceded by the
;; outcome of the game, which is 'MAX-WINS, 'MIN-WINS,
;; or 'DRAW.
;;
;; After this function is called, the global variable
;; *VISITS* holds the number of states visited while
;; the game was being played.
;;
(defun play-game (state)
  (setf *visits* 0)
  (let* ((value-and-moves
          (cond
           ((eql (state-next-player state) 'max)
            (find-value-and-moves state #'< #'>
                                  *state-min-value*
                                  *state-max-value*))
           (t
            (find-value-and-moves state #'> #'<
                                  *state-max-value*
                                  *state-min-value*))))
         (value (car value-and-moves))
         (moves (cdr value-and-moves)))
    (cond
     ((<= value *state-min-value*)
      (cons 'min-wins moves))
     ((>= value *state-max-value*)
      (cons 'max-wins moves))
     (t
      (cons 'draw moves)))))
```

Figure 3.22: Play-Game

```
;;;; Tic Tac Toe
;;;;
;;;; File:         tictactoe.in
;;;; Author:       <your name> (<your e-mail address>)
;;;; Version:      2

;; YOU ONLY NEED CHANGE THE LINE MARKED WITH ‘CHANGE’.

(load "game-search.lsp")
--->
;; Loading file game-search.lsp ...
;; Loading of file game-search.lsp is finished.
T

;; state := (number-of-moves-made . board )
;;
;; number-of-moves-made ::= 0 | 1 | ... | 9
;;      [This = number X’s in board
;;              + number of O’s in board]
;;
;; board := ((p00 p01 p02) (p10 p11 p12) (p20 p21 p22))
;;
;;              where board situation is    p00 p01 p02
;;                                          p10 p11 p12
;;                                          p20 p21 p22
;;
;; pij ::= X | O | _
;;
;;
;; move ::= (i . j)  moves to position pij
;;
;; i, j ::= 0 | 1 | 2

(defun number-of-moves-made (state) (car state))
--->
NUMBER-OF-MOVES-MADE
(defun board (state) (cdr state))
--->
BOARD
```

Figure 3.23: Tic Tac Toe Data: Page 1 of 2

```
(defvar *state-min-value* -1)
--->
*STATE-MIN-VALUE*
(defvar *state-max-value* +1)
--->
*STATE-MAX-VALUE*


;; Next player to move from given state, as 'X or 'O.
;;
(defun next-player (state)
  (cond
   ((evenp (number-of-moves-made state))
    'X)
   (t
    'O)))
--->
NEXT-PLAYER

;; Next player to move from given state, as 'MAX (X)
;; or 'MIN (O).
;;
(defun state-next-player (state)
  (cond
   ((evenp (number-of-moves-made state))
    'max)
   (t
    'min)))
--->
STATE-NEXT-PLAYER
```

Figure 3.24: Tic Tac Toe Data: Page 2 of 2

```
;; A ‘pattern’ has the same structure as a board and
;; contains NIL and T atoms that select squares of the
;; board.  Thus the pattern
;;      ’((T T T) (NIL NIL NIL) (NIL NIL NIL))
;; selects the squares of the first row of the board.
;;
;; This function returns the number of X’s in selected
;; squares minus the number of O’s in selected squares.
;; If there are 3 squares selected, +3 is returned if
;; the selected squares are all X’s and -3 if they are
;; all O’s, and some other number is returned in any
;; other situation.
;;
;; The function works for any two s-expressions,
;; PATTERN and BOARD, that have the same list structure,
;; and hence the function can be called recursively.
;;
;; RR: (count-moves nil ’a) ===> 0      a is an atom
;;     (count-moves t ’_) ===> 0
;;     (count-moves t ’X) ===> +1
;;     (count-moves t ’O) ===> -1
;;     (count-moves ’(r . s) ’(y . z)) ===>
;;              (count-moves ’r ’y)
;;              + (count-moves ’s ’z)
;;                      r, s, y, z are s-expressions
;;
```

Figure 3.25: Tic Tac Toe `Count-Moves` Rewrite Rules

```
(defun count-moves (pattern board)
  (cond
   ((not pattern)
    0)
   ((eql '_ board)
    0)
   ((eql 'X board)
    +1)
   ((eql 'O board)
    -1)
   ((consp pattern)
    (+ (count-moves (car pattern) (car board))
       (count-moves (cdr pattern) (cdr board))))))
--->
COUNT-MOVES



;; Patterns to select all the 3-in-a-line sets of
;; squares on the board.  E.g. all 3 rows,
;; all 3 columns, and the 2 diagonals.
;;
(defvar *winning-patterns*
  '(    ((t t t) (nil nil nil) (nil nil nil))
        ((nil nil nil) (t t t) (nil nil nil))
        ((nil nil nil) (nil nil nil) (t t t))
        ((t nil nil) (t nil nil) (t nil nil))
        ((nil t nil) (nil t nil) (nil t nil))
        ((nil nil t) (nil nil t) (nil nil t))
        ((t nil nil) (nil t nil) (nil nil t))
        ((nil nil t) (nil t nil) (t nil nil))
    ))
--->
*WINNING-PATTERNS*
```

Figure 3.26: Tic Tac Toe `Count-Moves` and `*Winning-Patterns*`

```
;; Return +1 if X wins, -1 if O wins, and 0 if there
;; are no moves left and neither X or O win.  Otherwise
;; return NIL.
;;
;; RR: (state-value 'state) ===>
;;           (state-value 'state *winning-patterns*)
;;      (state-value 'state '(pattern . other-patterns))
;;           ===> +1 if (count-moves
;;                         'pattern (board 'state)) = +3
;;           ===> -1 if (count-moves
;;                         'pattern (board 'state)) = -3
;;           ===> (state-value 'state 'other-patterns)
;;                                otherwise
;;      (state-value 'state '())
;;           ===> 0 if (number-of-moves-made 'state) = 9
;;           ===> nil otherwise
;;
(defun state-value
      (state &optional
             (pattern-list *winning-patterns*))
  (cond
   ((null pattern-list)
    (cond
     ((= 9 (number-of-moves-made state))
      0)
     (t
      nil)))
   (t
    (let ((value (count-moves (first pattern-list)
                              (board state))))
     (cond
      ((= +3 value)
       +1)
      ((= -3 value)
       -1)
      (t
       (state-value state (rest pattern-list))))))))
--->
STATE-VALUE
```

Figure 3.27: Tic Tac Toe `State-Value`

```
;; Return a new list equal to the given list but with
;; the n'th element of the list replaced by the give
;; value.  The list elements are numbered 0, 1, 2,
;; ... (length list) - 1.
;;
;; 0 <= n < (length list) is assumed.
;;
;; RR: (replace-nth 0 '(x . y) 'value) ===> '(value . y)
;;     (replace-nth n '(x . y) 'value) ===>
;;          (cons 'x (replace-nth ( (- n 1) 'y 'value)))
;;                                                   if n > 0
;;
;;      n is an integer, value, x, y are s-expressions
;;
(defun replace-nth (n list value)
  (cond
   ((= n 0)
    (cons value (rest list)))
   (t
    (cons (first list)
          (replace-nth (- n 1) (rest list) value)))))
--->
REPLACE-NTH

(defun state-next (state move)
  (cons
   (+ (number-of-moves-made state) 1)
   (replace-nth (car move) (board state)
                (replace-nth (cdr move)
                             (nth (car move)
                                  (board state))
                             (next-player state)))))
--->
STATE-NEXT
```

Figure 3.28: Tic Tac Toe State-Next

```
;; STATE-LEGAL-MOVES returns a list of the board
;; positions which contain '_'.  E.g. if the center
;; contains '_', then (1 . 1) is an element of the
;; list returned.
;;
;; The recursion variables are:
;;
;;      list of rows beginning with the i'th row
;;      list of positions in the i'th row, beginning
;;              with the j'th position in that row
;;              (position ::= X | O | _)
;;      i
;;      j
;;      result to be returns as computed so far
;;
;;
```

Figure 3.29: Tic Tac Toe `State-Legal-Moves`: Page 1 of 3

```
;; RR: (state-legal-moves 'state) ===>
;;           (state-legal-moves 'state (board 'state)
;;                              (first (board 'state))
;;                              0 0 nil)
;;
;;     (state-legal-moves 'state 'rows 'positions
;;                        3 j 'result)
;;          ===> 'result
;;     (state-legal-moves 'state 'rows 'positions
;;                        i 3 'result)
;;          ===> (state-legal-moves 'state (rest 'rows)
;;                                  (first (rest 'rows))
;;                                  (+ i 1) 0 'result)
;;                 if i < 3
;;     (state-legal-moves 'state 'rows 'positions
;;                        i 3 'result)
;;          ===> (state-legal-moves 'state 'rows
;;                                  (rest 'positions)
;;                                  i (+ j 1)
;;                                  '((i . j) . result))
;;                 if (eql (first 'positions) '_)
;;                    and i, j < 3
;;          ===> (state-legal-moves 'state 'rows
;;                                  (rest 'positions)
;;                                  i (+ j 1)
;;                                  'result)
;;                 if not (eql (first 'positions) '_)
;;                    and i, j < 3
;;
```

Figure 3.30: Tic Tac Toe `State-Legal-Moves`: Page 2 of 3

```
(defun state-legal-moves
      (state &optional (rows (board state))
             (positions (first rows))
             (i 0) (j 0)
             (result nil))
  (cond
   ((= i 3)
    result)
   ((= j 3)
    (state-legal-moves state (rest rows)
                       (first (rest rows))
                       (+ i 1) 0 result))
   ((eql '_ (first positions))
    (state-legal-moves state rows (rest positions)
                       i (+ j 1)
                       (cons (cons i j) result)))
   (t
    (state-legal-moves state rows (rest positions)
                       i (+ j 1) result))))
--->
STATE-LEGAL-MOVES
```

Figure 3.31: Tic Tac Toe `State-Legal-Moves`: Page 3 of 3

```
;; Print row as 3 characters using PRINC.
;;
(defun print-row (row)
  (cond
   ((null row))
   (t
    (princ (first row))
    (print-row (rest row)))))
--->
PRINT-ROW

;; Print state as per the following example:
;;
;;      Next Player is X
;;              X__
;;              O_X
;;              O__
;;
(defun state-print (state)
  (princ '|Next Player is |)
  (princ (next-player state))
  (fresh-line)
  (princ '|        |)
  (print-row (first (board state)))
  (terpri)
  (princ '|        |)
  (print-row (second (board state)))
  (terpri)
  (princ '|        |)
  (print-row (third (board state)))
  (terpri))
--->
STATE-PRINT
```

Figure 3.32: Tic Tac Toe `Print-State`

```
;; Tests of the STATE- functions:

(defvar *state-1* '(6 (_ X _) (O O X) (X _ O)))
--->
*STATE-1*

(state-next-player *state-1*)
--->
MAX

(state-print *state-1*)
--->
Next Player is X
        _X_
        OOX
        X_O
NIL

(state-legal-moves *state-1*)
--->
((2 . 1) (0 . 2) (0 . 0))

(state-value *state-1*)
--->
NIL

(state-value '(5 (X _ O) (O X _) (_ _ X)))
--->
1

(state-value '(9 (X X O) (O O X) (X X O)))
--->
0

(state-next *state-1* '(2 . 1))
--->
(7 (_ X _) (O O X) (X X O))
```

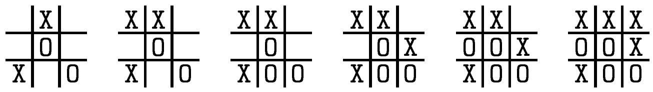
Figure 3.33: Tic Tac Toe State Function Tests

3.2.7 TicTacToe Results

Typical results for the above code are:

```
(defvar *state-2* '(4 (_ X _) (_ O _) (X _ O)))
=>
*STATE-2*

(play-game *state-2*)
=>
(MAX-WINS (0 . 0) (2 . 1) (1 . 2) (1 . 0) (0 . 2))

*visits*
=>
138
```

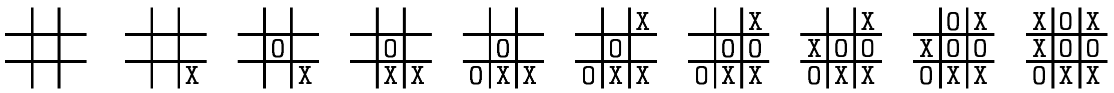
Pictorially the sequence of moves returned is:



People find this sequence of moves surprising. Instead of blocking `X`, `O`'s first move seems random. Instead of taking a win as soon as possible, `X` seems to take a random second move.

But it is still true that these moves lead to an optimal result, if you consider that only the final outcome counts, and not the length of the game. By the time of `O`'s first move, `O` has already lost the game, so all moves are equally optimal. By the time of `X`'s second move, any move will eventually win the game, so any move is optimal. Neither of the seemingly random moves are forced in any way.

The following is a full game played from an initially empty board:

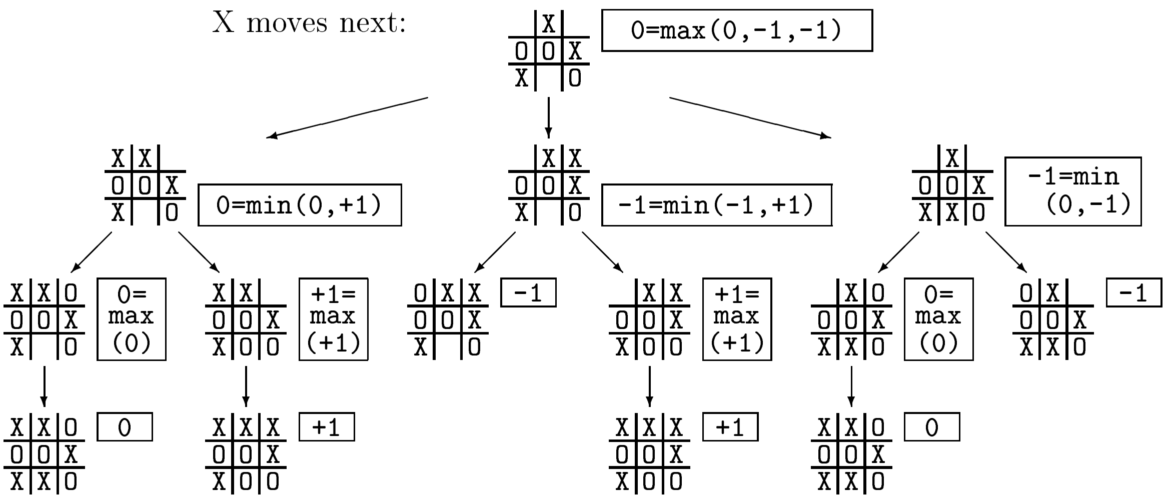


549,946 59,705 7,332 935 198 47 14 5 2 1

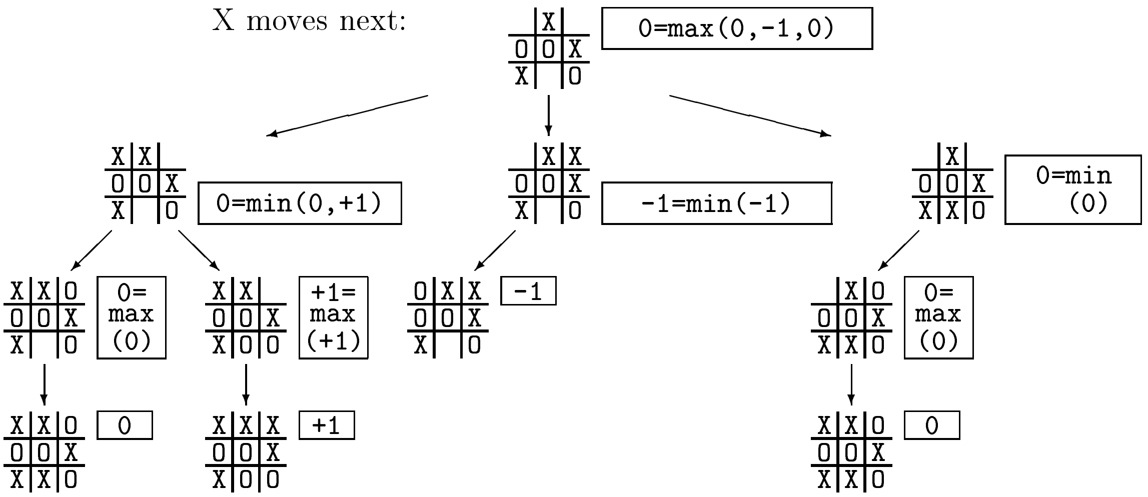
Under each board we have written the number of visits required by the `find-value-and-moves` function to discover the next move from that board. The number for a nearly empty board is very high, and can require a lot of computer time.

3.2.8 Alpha-Beta (Worst-Best) Pruning

This last result leads to a question: can we optimize the minimax algorithm. Consider the game search tree:



We can prune this tree to become:



The idea behind this pruning is that the MAX player (X) at the tree root is trying moves from left to right searching for a maximizing move. After MAX finds the first move with value 0, `MAX` can try the second move and tell the `MIN` player that `MAX` is no longer interested in values less than 0, as these would not be maximum in the presence of the value 0. So as soon as `MIN` makes a move with value 0 or less, `MIN` can stop, since `MIN` cannot produce any value greater than 0 once `MIN` has found a move with value 0.

More precisely, each player is asked to produce a value in a range of possible values. What this means is that if a player finds a value equal to or better than the best value in the range for that player, the player should stop making moves. Initially `MAX` is asked to produce a value in the range [-1,+1], but after `MAX` finds the 0 value, `MAX`'s range of interest changes to [0,+1], and thereafter `MIN` only needs to search in this range.

The **`find-value`** function described at the end of Section 3.2.3 can be upgraded to take two extra arguments, '*worst*' and '*best*', that define the range of values of interest. The pseudocode for the upgraded function becomes:

Pseudocode for `(find-value '`*state* `#'`*worse* `#'`*better* `'`*worst* `'`*best*`)`:

1. If *state* can be given a direct value, return that value.
2. For each legal move from *state*:
   (a) Compute the next state s reached by the move.
   (b) Compute a value for s by calling:

   `(find-value '`*s* `#'`*better* `#'`*worse* `'`*best* `'`*worst*`)`:

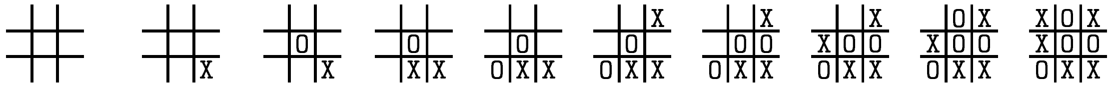
   (c) If value >= *best*, return value.
   (d) If value > *worst*, set *worst* = value.
3. Return *worst* (the best of the values).

When enhancing this function to return a list of moves as well as a value, one must be careful. If the recursive call in Step 2b returns the value *worst*, that may not be the true value, and the true value may be even worse. Thus if *worst* is 0 when the recursive call is made, the next player is `MIN`, and `MIN` returns 0, the true value may be -1, and `MAX` should not make the corresponding move expecting a draw. The correct move to make is the one associated with the first move that produces the value finally returned in Step 3, as this move will have been evaluated when *worst* was less than the value found.

Historically 'worst' is called '**alpha**' and 'best' is called '**beta**'. Because the effect of using alpha and beta is to prune the game search tree, this optimization is called '**alpha-beta pruning**'.

3.2.9 Results for Alpha-Beta Pruning

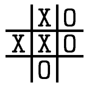
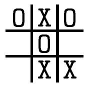
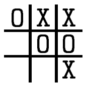
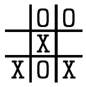
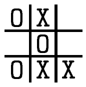
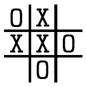
The following is a full game played from an initially empty board, where we have written under each board the number of visits required to discover the next move with alpha-beta pruning. For reference we have also written the number of visits required without alpha-beta pruning in square brackets [...]. The improvement for an empty board is a factor of 30, and ditto for a board with one non-empty square.



16,811	1,903	728	61	50	17	10	5	2	1
[549,946]	[59,705]	[7,332]	[935]	[198]	[47]	[14]	[5]	[2]	[1]

Exercise 3.2.9.1 For each of the following game states (for all of which X moves next):

(A) Draw the game tree whose root is the given state.
(B) For each node with with a directed value, label the node with the directed value in a rectangle. Circle any subtree pruned from the game tree by not going below states with directed values.
(C) Label each remaining node with its minimax value.
(D) For each node write the alpha and beta values effective when the node is examined, labeling the nodes with these values as an ordered pair '$[\alpha, \beta]$'. Give the root node the label $[-1, 1]$. Leave this label off any node that would be not visited because of alpha-beta pruning.

(a)  (b)  (c)  (d)  (e)  (f) 

3.3 Resolution Theorem Proving

Previously we have given two ways of proving theorems in propositional calculus, but both methods tend to be $O(2^N)$ in the number of variables N in the theorem. Here we will give the '**resolution**' method that runs much faster on typical problems. We will cast the resolution method as a goal search, though there are some minor complications because the resolution method naturally searches a hypergraph and not an ordinary directed graph.

3.3.1 Conjunctive Normal Form

The resolution method requires that the negation of the theorem to be proved be put into conjunctive normal form.

A wff w is said to be in **conjunctive normal form** (**cnf**) if both the following are true:

- All $\wedge$'s in w are outside all $\vee$'s and $\neg$'s.
- All $\neg$'s in w are inside all $\wedge$'s and $\vee$'s and are not doubled.

Cnf formulae can be defined by the following grammar:

$$
\begin{array}{lcl}
\text{literal} & ::= & \text{variable} \mid (\neg\ \text{variable}) \\
\text{cnf-clause} & ::= & \text{literal} \mid (\text{cnf-clause} \vee \text{cnf-clause}) \\
\text{cnf} & ::= & \text{cnf-clause} \mid (\text{cnf} \wedge \text{cnf})
\end{array}
$$

We can use the standard equivalences to rewrite any wff w to a cnf formula that is equivalent to w. An example is:

$$
\begin{array}{ll}
 & (\neg(p\wedge(q\vee(\neg p)))) \\
\Longrightarrow & ((\neg p)\vee(\neg(q\vee(\neg p)))) \\
\Longrightarrow & ((\neg p)\vee((\neg q)\wedge(\neg(\neg p)))) \\
\Longrightarrow & ((\neg p)\vee((\neg q)\wedge p)) \\
\Longrightarrow & (((\neg p)\vee(\neg q))\wedge((\neg p)\vee p))
\end{array}
$$

3.3.2 Exponentially Exploding Theorem Provers

Consider trying to prove the following theorem:

$$w \;\equiv\; ((p_1 \wedge (p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_3)) \Rightarrow p_3)$$

Although this is a simple theorem, it has a structure typical of theorems people really want to prove, which tend to consist of the conjunction of implications.

Recalling that:

$$\begin{aligned} (p \Rightarrow q) \;&\equiv\; ((\neg p) \vee q) \\ (\neg(p \Rightarrow q)) \;&\equiv\; (p \wedge (\neg q)) \end{aligned}$$

we can compute:

$$\begin{aligned} (\neg w) \;&\equiv\; (p_1 \wedge (p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_3) \wedge (\neg p_3)) \\ &\equiv\; (p_1 \wedge ((\neg p_1) \vee p_2) \wedge ((\neg p_2) \vee p_3) \wedge (\neg p_3)) \quad \text{[CNF]} \\ &\equiv\; ((p_1 \wedge (\neg p_1) \wedge (\neg p_2) \wedge (\neg p_3)) \vee \ldots) \qquad \text{[DNF]} \end{aligned}$$

If instead of 3 p's there are N p's, the truth table of w has 2^N rows, and the disjunctive normal form of $(\neg w)$ has 2^{N-1} dnf-clauses.[2]

But the conjunctive normal form of $(\neg w)$ has only $N+1$ cnf-clauses!

So if we could find a reasonably fast way of proving that a formula in CNF form was unsatisfiable, we could prove typical theorems efficiently.

3.3.3 Resolutions

Resolution is a method for proving that a CNF formula is unsatisfiable that is based on the following:

Lemma: If p, $q \in$ wff; $v \in$ variable, then: $\quad ((p \vee v) \wedge (q \vee (\neg v))) \Rightarrow (p \vee q)$

[2] Actually in the example we give one can use special rules to simplify the dnf as it is being computed, e.g., by reducing the logical 'and' of a variable and its negation to false, and thereby avoid the exponential explosion. But with more complex implications such simplification is either not possible or not known.

Proof: If v is true, then q is true, hence $(p \vee q)$ is true.
If v is false, then p is true, hence $(p \vee q)$ is true.

$(p \vee q)$ is called the resolution of the two cnf-clauses: $(p \vee v)$ and $(q \vee (\neg v))$.

Here p may be missing: $(v \wedge (q \vee (\neg v))) \Rightarrow q$.

Alternatively q may be missing: $((p \vee v) \wedge (\neg v)) \Rightarrow p$.

Or both p and q may be missing: $(v \wedge (\neg v)) \Rightarrow \mathbf{f}$.

Exercise 3.3.3.1 For each of the following clause pairs, find all the resolutions of the pair (there may be zero, and there may be more than one if there is more than one variable that can be cancelled):

(a) $\{p,\ q\}$ and $\{r,\ (\neg q),\ s\}$
(b) $\{r,\ s,\ (\neg v)\}$ and $\{(\neg s),\ (\neg v),\ s\}$
(c) $\{(\neg m),\ n,\ (\neg p)\}$ and $\{(\neg s),\ (\neg v),\ s\}$
(d) $\{(\neg m),\ n,\ (\neg p)\}$ and $\{(\neg m),\ (\neg n),\ p\}$
(e) $\{r,\ (\neg s),\ v,\ (\neg w)\}$ and $\{(\neg r),\ (\neg s),\ v\}$

3.3.4 The Resolution Hypergraph

Suppose we denote a **CNF-clause** by the set of literals it contains. For example, $(p_1 \wedge (\neg p_2) \wedge p_3)$ is denoted by $\{p_1, (\neg p_2), p_3\}$. With this notation, the definition of resolution becomes:

The **resolution** of two CNF-clauses $P \cup \{v\}$ and $Q \cup \{(\neg v)\}$ is $P \cup Q$, the set of all the literals in the two clauses but with v removed from $P \cup \{v\}$ and $(\neg v)$ removed form $Q \cup \{(\neg v)\}$. Here P is a set of literals not containing v, and Q is a set of literals not containing $(\neg v)$. (But it is possible for P and hence $P \cup Q$ to contain $(\neg v)$, or for Q and hence $P \cup Q$ to contain v.)

By the lemma of the previous section, if two clauses can be resolved, then the conjunction of the two clauses implies their resolution.

We now form a '**hypergraph**' whose nodes are sets of literals denoting CNF-clauses, and whose hyperedges correspond to resolutions. If nodes P and Q resolve to R, $(\{P, Q\}, R)$ is a hyperedge, and is represented by drawing two arrows, one from P to R, and another from Q to R, and connecting these arrows with a line (the line is often omitted).

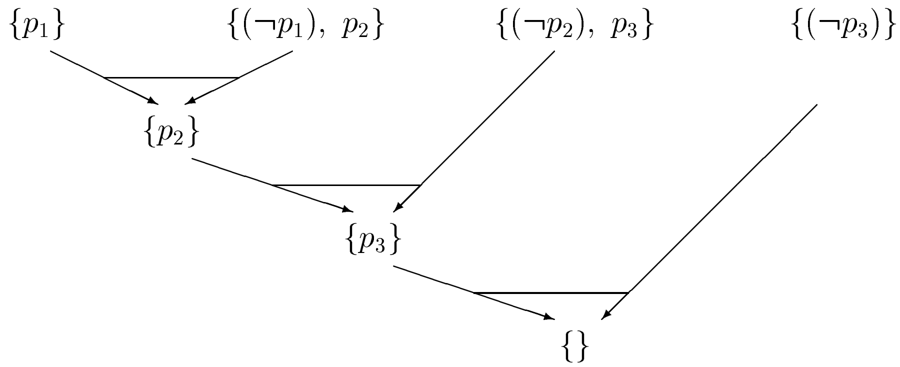
For example, given the formula:

$$w \equiv ((p_1 \wedge (p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_3)) \Rightarrow p_3)$$

its negation is:

$$(\neg w) \equiv (p_1 \wedge ((\neg p_1) \vee p_2) \wedge ((\neg p_2) \vee p_3) \wedge (\neg p_3))$$

and the resolution hypergraph that is 'rooted' at the 4 clauses of $(\neg w)$ is:



In this graph **f** is denoted by $\{\}$, and is in the graph because $(p_3 \wedge (\neg p_3)) \Rightarrow \mathbf{f}$.

Because the parents of each child in this hypergraph imply the child, and $(\neg w)$ implies the root clauses at the top of the hypergraph, it follows that $(\neg w) \Rightarrow \mathbf{f}$, and therefore $(\neg w)$ is unsatisfiable, and w is a theorem.

Exercise 3.3.4.1 For each of the following sets of clauses, draw the complete resolution hypergraph rooted at the set of clauses. No two nodes in this hypergraph should be labeled with the same clause.

(a) $\{p,\ q\},\quad \{r,\ (\neg q),\ s\},\quad \{(\neg p)\}$
(b) $\{(\neg r)\},\quad \{r,\ s,\ (\neg v)\},\quad \{(\neg s),\ (\neg v),\ s\}$
(c) $\{(\neg m),\ n,\ (\neg p)\},\quad \{(\neg n),\ p\}$
(d) $\{r,\ (\neg s),\ v,\ (\neg w)\},\quad \{(\neg r),\ s,\ v\}$

3.3.5 The Closure of a Clause Set

Given a set of clauses (e.g. all clauses of a CNF formula), we define its **closure** as follows:

1. Find all resolutions of pairs of clauses in the clause set.
2. Add these resolutions to the clause set.
3. Repeat (1) and (2) until there are no new resolutions.

Lemma: If all the clauses in a clause set are true for some assignment, then all the clauses in the closure of the clause set are true for that assignment.

Corollary: If the closure of a clause set contains the empty clause, then for each assignment some member of the clause set is unsatisfiable.

Corollary: If the closure of the clause set of $(\neg w)$ contains the empty clause, then w is a theorem.

Proof: $(\neg w) \Rightarrow \mathbf{f}$ so $(\neg w) \equiv \mathbf{f}$.

Resolution Theorem: The converse to the above is also true, so that w is a theorem if and only if the closure of the clause set of the CNF of $(\neg w)$ contains the empty clause.

Proof: Given in another book ([LP81]).

Thus we are lead to:

Automatic Theorem Proving Algorithm 3: Compute the CNF of $(\neg w)$ and search for the empty clause in the closure of the CNF's clause set.

Exercise 3.3.5.1 For each of the following sets of clauses, compute the closure of the clause set. Write this closure as a list of clauses, one per line, and for each clause derived by resolution, indicate the clauses that were resolved to get it.

(a) $\{p,\ (\neg q)\},\quad \{q,\ (\neg r)\},\quad \{r,\ (\neg s)\}$
(b) $\{p, q, r, t\},\quad \{(\neg q),\ r\},\quad \{r,\ (\neg t)\}$
(c) $\{p,\ (\neg q),\ r\},\quad \{q,\ s,\ t\},\quad \{p,\ (\neg s),\ t\}$
(c) $\{p,\ q\},\quad \{r,\ (\neg q),\ s\},\quad \{(\neg r),\ (\neg q),\ s\}$

Exercise 3.3.5.2 Apply the method of automatic theorem proving algorithm 3 to decide whether the following wff are theorems. Carefully show rewrites using the cnf rewrite rules. Carefully compute closures as in the previous exercise, though you may stop computing the closure if you get to the empty clause, and need not compute the whole closure in this case.

(a) $((\neg\ ((p\ \vee\ (\neg\ q))\ \wedge\ q))\ \vee\ p)$
(b) $((\neg\ (((\neg\ p)\ \vee\ (\neg\ q))\ \wedge\ q))\ \vee\ p)$
(c) $((p\ \vee\ q)\ \vee\ ((\neg\ p)\ \wedge\ (\neg\ q)))$
(d) $((p\ \vee\ q)\ \vee\ (p\ \wedge\ (\neg\ q)))$
(e) $((\neg\ (((\neg\ p)\ \vee\ \mathtt{q})\ \wedge\ (((\neg\ q)\ \vee\ r)\ \wedge\ p)))\ \vee\ r)$

3.3.6 Two Out of Three Example

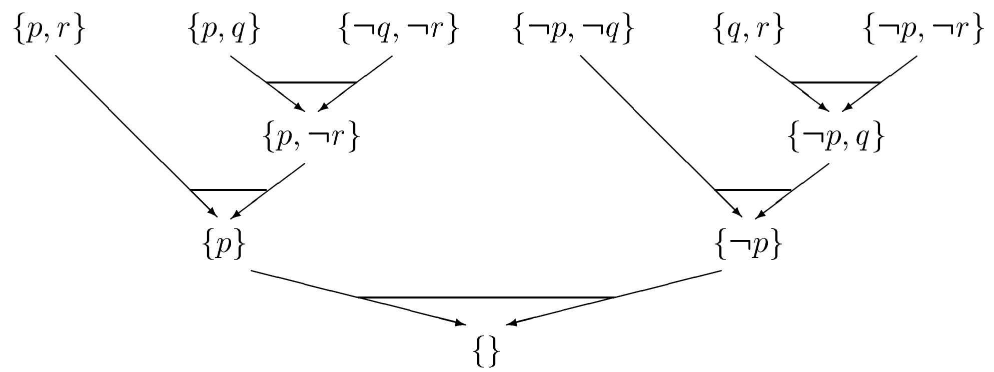
The **two out of three** example is the propositional calculus equivalent of "Either 2 out of 3 are true or 2 out of 3 are false". This can be rendered as the wff:

$$w \ \equiv\ ((p\wedge q)\vee(q\wedge r)\vee(r\wedge p)\vee((\neg p)\wedge(\neg q))\vee((\neg q)\wedge(\neg r))\vee((\neg r)\wedge(\neg p)))$$

whose negation in CNF form is:

$$(\neg w)\ \equiv\ (((\neg p)\vee(\neg q))\wedge((\neg q)\vee(\neg r))\wedge((\neg r)\vee(\neg p))\wedge(p\vee q)\wedge(q\vee r)\wedge(r\vee p))$$

Part of the resolution hypergraph of this negation is:



Since $\{\}$ appears in this hypergraph, w is a theorem.

3.3.7 Search Algorithm

Searching a resolution hypergraph for the goal node '$\{\}$' can be implemented by using our breadth first or depth first goal search engines and a trick.

First, we define a state to be a clause.

Next, we define the `goal` function to be true of the empty clause.

And now the trick. We define the `children` function to return the resolutions of the state s with any already seen state that resolves with s. In effect, each hyperarc is converted into an ordinary arc where the parent is the node being visited and the other hyperarc parent, which is to be omitted, is a clause on the list of seen states. This works because all we need to do is find the hyperarcs whose source is any pair of visited states, but when the second state is visited, the first state will be in the list of seen states.

Parts of an implementation of the resolution data abstraction is given in Figures 3.34 through 3.37. The `children` function is omitted and left as an exercise for the reader. Also omitted for the same reason is a `resolve` function that takes two clauses and a literal in each clause and produces the union of the clauses minus their respective literals.

The `resolve` function should print the two input clauses and the resolution of these inputs if the global variable `*resolution-trace*` is true.

Results for the `two out of three` theorem are given in Figures 3.38 and 3.39. For use in testing the more complex `pigeon hole` theorem is given in Figure 3.40 without results.

The results given in Figures 3.38 and 3.39 use a `children` function that has two optimizations built into it:

- Any clause that contains both v and $(\neg v)$ for some variable v is not considered to be a child. Such clauses are called '**useless clauses**'.
- Any clause that is a superset of some already seen clause is not considered to be a child.

There is code in Figures 3.36 and 3.37. to support these optimizations.

The following can be proved:

Lemma: All clauses generated by the non-optimized algorithm are either supersets of clauses generated by the optimized algorithm, or are useless clauses.

Corollary: The optimized algorithm generates the empty clause if and only if the non-optimized algorithm does.

```
;;;; Resolution Theorem Proving
;;;;
;;;; File:         resolution.in
;;;; Author:       <your name> (<your e-mail address>)
;;;; Version:      1

(load "search.lsp")
--->
;; Loading file search.lsp ...
;; Loading of file search.lsp is finished.
T

(defvar *trace-resolutions* nil)
--->
*TRACE-RESOLUTIONS*

;; Data Structure;
;;
;; state ::= clause
;;
;; clause ::= () | ( literal . clause )
;;
;; literal ::= variable | (NOT variable)
;;
;; variable ::= symbol
;;
;; The literals in a clause are sorted by the variables
;; they contain, with any negative literal appearing
;; immediately after any positive literal with
;; the same variable.
;;
;; Duplicate literals are eliminated from the clause.

(defun goal (state) (null state))
--->
GOAL
```

Figure 3.34: Resolution Data

```
;; Function to compare to literals.  The first
;; is less than the second if the first's symbol
;; is alphabetically less than the second's symbol,
;; or if the first literal is positive and the
;; second literal is negative.
;;
;; RR: (<< 's1 's2) ===> s1 < s2
;;     (<< '(not s1) 's2) ===> s1 < s2
;;     (<< 's1 '(not s2)) ===> s1 <= s2
;;     (<< '(not s1) '(not s2)) ===> s1 < s2
;;
;;     s1, s2 are symbols
;;
;; where s1 < s2 means that the symbol s1 is
;; alphabetically less than s2 (using string-lessp).
;;
(defun << (literal1 literal2)
  (cond
   ((symbolp literal1)
    (cond
     ((symbolp literal2)
      (string-lessp literal1 literal2))
     (t
      (or (eql literal1 (second literal2))
          (string-lessp literal1 (second literal2))))))
   (t
    (cond
     ((symbolp literal2)
      (string-lessp (second literal1) literal2))
     (t
      (string-lessp (second literal1)
                    (second literal2)))))))
--->
<<
```

Figure 3.35: Literal << Function

```
;; Given a literal return its negation.
;;
;; RR: (negate 's) ===> '(not s)
;;     (negate '(not s)) ===> 's
;;
;;     s is a symbol
;;
(defun negate (literal)
  (cond
   ((symbolp literal)
    `(not ,literal))
   (t
    (second literal))))
--->
NEGATE

;; Function to check whether clause1 contains all the
;; literals of clause2: i.e. is clause1 a superclause
;; of clause2.  Returns true if yes and false if no.
;;
;; RR: (superclause 'clause1 '()) ===> true
;;     (superclause '() '(y . z)) ===> false
;;     (superclause '(w . x) '(y . z))
;;         ===> (superclause 'x 'z)       if (equal 'w
;;                                                    'y)
;;         ===> (superclause 'x '(y .z))  otherwise
;;
(defun superclause (clause1 clause2)
  (cond
   ((null clause2) t)
   ((null clause1) nil)
   ((equal (first clause1) (first clause2))
    (superclause (rest clause1) (rest clause2)))
   (t
    (superclause (rest clause1) clause2))))
--->
SUPERCLAUSE
```

Figure 3.36: Literal Negate and Superclause

```
;; Return true if a clause contains some variable and
;; also its negation.
;;
;; RR: (useless-clause '()) ===> false
;;     (useless-clause '(x)) ===> false
;;     (useless-clause '(s (not s) . y)) ===> true
;;     (useless-clause '(x . y))
;;          ===> (useless-clause 'y) if the above are
;;                                    not applicable
;;
;;     s is a symbol, x is a literal, y is a clause
;;
(defun useless-clause (clause)
  (cond
   ((null clause)
    nil)
   ((null (rest clause))
    nil)
   ((and (symbolp (first clause))
         (consp (second clause))
         (eql (first clause) (second (second clause))))
    t)
   (t
    (useless-clause (rest clause)))))
--->
USELESS-CLAUSE
```

Figure 3.37: `Useless-Clause`

```
;; Either two out of three are true or
;; two out of three are false.

(defvar *start-states-3*
  '((a b) (b c) (a c)
    ((not a) (not b))
    ((not b) (not c))
    ((not a) (not c))))
--->
*START-STATES-3*

(breadth-first-search *start-states-3*
  #'children #'goal)
--->
Resolving (A B) and ((NOT B) (NOT C)) to
    (A (NOT C))
Resolving (A B) and ((NOT A) (NOT C)) to
    (B (NOT C))
Resolving (B C) and ((NOT A) (NOT B)) to
    ((NOT A) C)
Resolving (B C) and ((NOT A) (NOT C)) to
    ((NOT A) B)
Resolving (B C) and (B (NOT C)) to
    (B)
Resolving (A C) and ((NOT A) (NOT B)) to
    ((NOT B) C)
Resolving (A C) and ((NOT B) (NOT C)) to
    (A (NOT B))
Resolving (A C) and (A (NOT C)) to
    (A)
Resolving (A C) and ((NOT A) C) to
    (C)
Resolving ((NOT A) (NOT B)) and (B) to
    ((NOT A))
Resolving ((NOT A) (NOT B)) and ((NOT A) B) to
    ((NOT A))
Resolving ((NOT A) (NOT B)) and (A) to
    ((NOT B))
```

Figure 3.38: Two Out of Three Example: Page 1 of 2

```
Resolving ((NOT A) (NOT B)) and (A (NOT B)) to
    ((NOT B))
Resolving ((NOT B) (NOT C)) and (B (NOT C)) to
    ((NOT C))
Resolving ((NOT B) (NOT C)) and (B) to
    ((NOT C))
Resolving (B) and ((NOT B)) to
    NIL
Resolving (B) and ((NOT B)) to
    NIL
(NIL (B) (B C))

(depth-first-search *start-states-3*
  #'children #'goal)
--->
Resolving (A B) and ((NOT B) (NOT C)) to
    (A (NOT C))
Resolving (A B) and ((NOT A) (NOT C)) to
    (B (NOT C))
Resolving (B (NOT C)) and (B C) to
    (B)
Resolving (B (NOT C)) and ((NOT B) (NOT C)) to
    ((NOT C))
Resolving ((NOT C)) and (A C) to
    (A)
Resolving (A) and ((NOT A) (NOT B)) to
    ((NOT B))
Resolving ((NOT B)) and (B C) to
    (C)
Resolving ((NOT B)) and (B) to
    NIL
(NIL ((NOT B)) (A) ((NOT C)) (B (NOT C)) (A B))
```

Figure 3.39: Two Out of Three Example: Page 2 of 2

```
;; Pigeonhole Principle: If there are three pigeons and
;; two pigeon holes and every pigeon is in a hole, then
;; some hole has at least two pigeons.
;;
;; pij means pigeon i is in hole j.
;;
;; (implies (and (or p11 p12)
;;               (and (or p21 p22)
;;                    (or p31 p32)))
;;          (or (and p11 p21)
;;              (or (and p21 p31)
;;                  (or (and p31 p11)
;;                      (or (and p12 p22)
;;                          (or (and p22 p32)
;;                              (and p32 p12)))))))
;; ==
;; (or (not (and (or p11 p12)
;;               (and (or p21 p22)
;;                    (or p31 p32))))
;;     (or (and p11 p21)
;;         (or (and p21 p31)
;;             (or (and p31 p11)
;;                 (or (and p12 p22)
;;                     (or (and p22 p32)
;;                         (and p32 p12)))))))
;;

(defvar *start-states-4*
  '((p11 p12) (p21 p22) (p31 p32)
    ((not p11) (not p21)) ((not p21) (not p31))
    ((not p11) (not p31)) ((not p12) (not p22))
    ((not p22) (not p32)) ((not p12) (not p32))))
--->
*START-STATES-4*
```

Figure 3.40: Pigeon Hole Example

3.4 Logic Programming

The **predicate calculus** extends the propositional calculus by permitting variables representing objects of the real world, such as people, places, numbers, etc., to occur in logical statements. In propositional calculus we used variables such as p and q to represent propositions such as 'Bob walked Sparky' and 'Sparky likes hamburger'. In predicate calculus it is common to represent propositions by **predicates** such as '`walked (Bob,Sparky)`' and '`likes (Sparky, hamburger)`'. Here '`Bob`', '`Sparky`', and '`hamburger`' are the names of constant objects. Variables can also be included in predicates, as in '`walked (Bob,`X`)`', where X is a variable.

Because we are going to program in LISP, we will represent predicates by arbitrary s-expressions, such as:

```
(bob walked sparky)
(sparky likes hamburger)
```

We will represent variables by quoted atoms, such as `'X` and `'WHO` in:

```
(bob walked 'X)
('WHO likes hamburger)
```

We will capitalize variable names to make them more visible inside predicates, but of course LISP does not distinguish between capital and non-capital letters. The choice of a single quote (`'`) to denote variables is somewhat of a malapropism, because in LISP the single quote denotes a constant as opposed to a variable, but the single quote is the best way to mark a predicate variable that is available to us in our limited subset of LISP.

Now suppose we have a database of predicates:

```
(fop is male)
(mop is female)
(dop is female)
(sop is male)
(sop is the son of fop)
(sop is the son of mop)
(dop is the daughter of fop)
(dop is the daughter of mop)
```

Given this database, suppose we pose the query:

```
('X is female)?
```

which is intended to find all the possible values of the variable `'X` for which the query is a predicate in the database. The query can be answered by a matching process that produces the two answers:

```
'X ===> mop
'X ===> dop
```

Another possible query would be:

```
('X is the son of 'Y)?
```

which produces the two answers:

```
'X ===> sop; 'Y ===> fop
'X ===> sop; 'Y ===> mop
```

And yet another possible query would be:

```
('X is the son of 'X)?
```

for which there are zero answers, as in this database no one is the son of himself.

Now suppose we want to pose the queries:

```
('PARENT is a parent)?
('FATHER is a father)?
```

A good way to answer these queries is to add the following to the database:

```
(('X is a parent) <=== ('Y is the son of 'X))
(('X is a parent) <=== ('Y is the daughter of 'X))
(('X is a father) <=== ('X is a parent) and ('X is male))
```

Here `<===` means 'is implied by'. Here also the variables `'X` and `'Y` are '**free variables**', in that any value (i.e., any s-expression) can be substituted for these variables. They are in contrast to the variables in queries, which are '**bound variables**', that have particular values that are not yet known.

So with these additions to the database and the proper program for searching for solutions to queries, the last two queries give the answers:

```
'PARENT ===> fop
'PARENT ===> mop
'FATHER ===> fop
```

In the rest of this section we will give a precise description of the algorithm used to search for solutions to queries of the sort just described.

Exercise 3.4.1 Consider the database:

```
(b to c)       (b to d)
(c to e)       (c to d)              (c to f)
(d to e)       (path 'X to 'X)
((path 'X to 'Z) <=== ('X to 'Y) and (path 'Y to 'Z))
```

Then what are the answers to the following queries:

(a)	`(path b to 'Y)?`	(d)	`(path 'X to d)?`
(b)	`(path c to 'Y)?`	(e)	`(path 'X to e)?`
(c)	`(path d to 'Y)?`	(f)	`(path 'X to f)?`

3.4.1 Logic Programs

A database of the above kind is called a **logic program**. It is a list of definite clauses made from predicates. A **predicate** is just an s-expression. Predicates can contain **variables**, that are quoted atoms. A **definite clause** consists of a head and a body. The **head** is just a predicate, and the **body** is a conjunctive list of predicates. The meaning of a definite clause is that the logical 'and' of all the predicates in the body implies the head.

A grammar for logic programs is given in Figure 3.41.

Note that a body can be empty, meaning the head is always true. In this case the `<===` can be omitted. An example of such a definite clause is '`((fop is male))`'.

```
predicate                ::= s-expression
variable                 ::= 'variable-name
variable-name            ::= atom except nil
definite-clause          ::= ( head <=== . body)
                           | ( head )
head                     ::= predicate
body                     ::= predicate-list
predicate-list           ::= () | non-empty-predicate-list
non-empty-predicate-list ::= ( predicate )
                           | ( predicate AND
                                    . non-empty-predicate-list )
database                 ::= ()
                           | ( definite-clause . database )
query                    ::= predicate-list
substitution             ::= ()
                           | ( ( variable-name s-expression )
                                    . substitution )
state                    ::= ( substitution . query )
```

Figure 3.41: Logic Program LISP Data Syntax

Note the doubled parentheses that are necessary when treating a predicate by itself as a definite clause and adding it to the database in LISP. When writing databases in text we often omit one pair of such doubled parentheses.

In general a **query** is just a list of predicates all of which must be simultaneously satisfied. The predicates may contain variables that are taken to be bound: that is, they represent particular but unknown values.

3.4.2 The SLD-Resolution Algorithm

The algorithm we describe that searches the database for answers to queries is called the **SLD-resolution algorithm**[3].

We will trace how this algorithm processes the query:

```
('X is a father)?
```

The SLD-resolution algorithm is a goal search algorithm. The state consists of a substitution and a query. A substitution is a map from variables to the expressions that are to replace the variables; some examples are:

$$\{\ \texttt{'X} \mapsto \texttt{sop},\ \texttt{'Y} \mapsto \texttt{fop}\ \}$$
$$\{\ \texttt{'PARENT} \mapsto \texttt{fop}\ \}$$

The substitution in the current state is to be applied to the query in the current state. Thus the state:

$$\{\ \texttt{'X} \mapsto \texttt{dop}\ \}\quad \texttt{('X is the daughter of 'Y)}$$

is essentially the same as the state:

$$\{\}\quad \texttt{(dop is the daughter of 'Y)}$$

which has made the substitution $\{\ \texttt{'X} \mapsto \texttt{dop}\ \}$ in the query and replaced the substitution by the empty substitution $\{\}$. The substitution part of the state is added to

[3] SLD stands for 'linear' resolution with 'definite clauses' and a 'selection function'.

by the matching process that discovers what expressions variables should be replaced by when it matches a query predicate with the head of a definite clause.

The initial state consists of the empty substitution and the initial query, so for our example the initial state is:

```
{}    ('X is a father)
```

The SLD-resolution algorithm must find the children of a state in the search tree. It does this by selecting the first predicate in the state's query[4], and finding all the definite clauses in the database whose heads match this predicate. For each match, the algorithm generates a child by replacing the selected predicate by the body of the definite clause whose head matched. During the match, variable mappings are added to the parent state substitution to make the substitution of the child.

In the case of our initial state, there is only one definite clause whose head might match the first (and only) predicate in the theory, because there is only one definite clause whose head contains the word '`father`'. This definite clause is:

```
(('X is a father) <=== ('X is a parent) and ('X is male))
```

The first problem is that the variable `'X` in this definite clause is free, and is not the same variable as the bound variable `'X` in the query predicate. Free variables may be replaced by any s-expression, but we do not know what s-expression to replace the free `'X` by. So instead we replace all the free variables in the definite clause by new bound variables that have not been used before. These bound variables stand for particular expressions to be determined later.

For convenience of coding, we use the integers `1`, `2`, `3`, ... as names for new bound variables. We simply keep a counter that we increment whenever we need a new bound variable name. We assume that query variables will not have integer names. So with this in mind we produce from the definite clause in the database a new definite clause:

```
(('1 is a father) <=== ('1 is a parent) and ('1 is male))
```

[4] This 'select the first predicate' operation is the 'selection function' referred to by the S in SLD. Another selection function might be used, but ours is the usual selection function.

where `'1` is a bound variable not used elsewhere.

Next we match the selected predicate, `('X is a father)`, to the head of the definite clause with bound variables, `('1 is a father)`. This match succeeds while adding `'X` $\mapsto$ `'1` to the parent state's substitution to make the child state's substitution. We then replace the selected predicate by the body of the definite clause with bound variables to make the child state's query. The body is the set of predicates that must be true in order for the selected predicate to be true according to the definite clause. So if we replace the selected predicate by the clause body, and if we can show that the new query is true for some values of the bound variables it contains, then the original query will be true for those variable values.

The child state of the initial state is therefore:

{ 'X $\mapsto$ '1 }   `('1 is a parent) and ('1 is male)`

This is the only child of the initial state, as only one definite clause head in the database matches the initial state query.

In computing the children of this new state, the SLD algorithm searches the database for definite clause head's that will match the state's first query predicate, `('1 is a parent)`. Here there are two such definite clauses:

```
(('X is a parent) <=== ('Y is the son of 'X))
(('X is a parent) <=== ('Y is the daughter of 'X))
```

Replacing the free variables in these definite clauses by new bound variables not previously used we get:

```
(('2 is a parent) <=== ('3 is the son of '2))
(('4 is a parent) <=== ('5 is the daughter of '4))
```

Matching `('1 is a parent)` to the head of the first of these definite clauses is successful and yields the state:

{ '1 $\mapsto$ '2, 'X $\mapsto$ '1 }   `('3 is the son of '2) and ('1 is male)`

while matching `('1 is a parent)` to the head of the second definite clause is successful and yields the state:

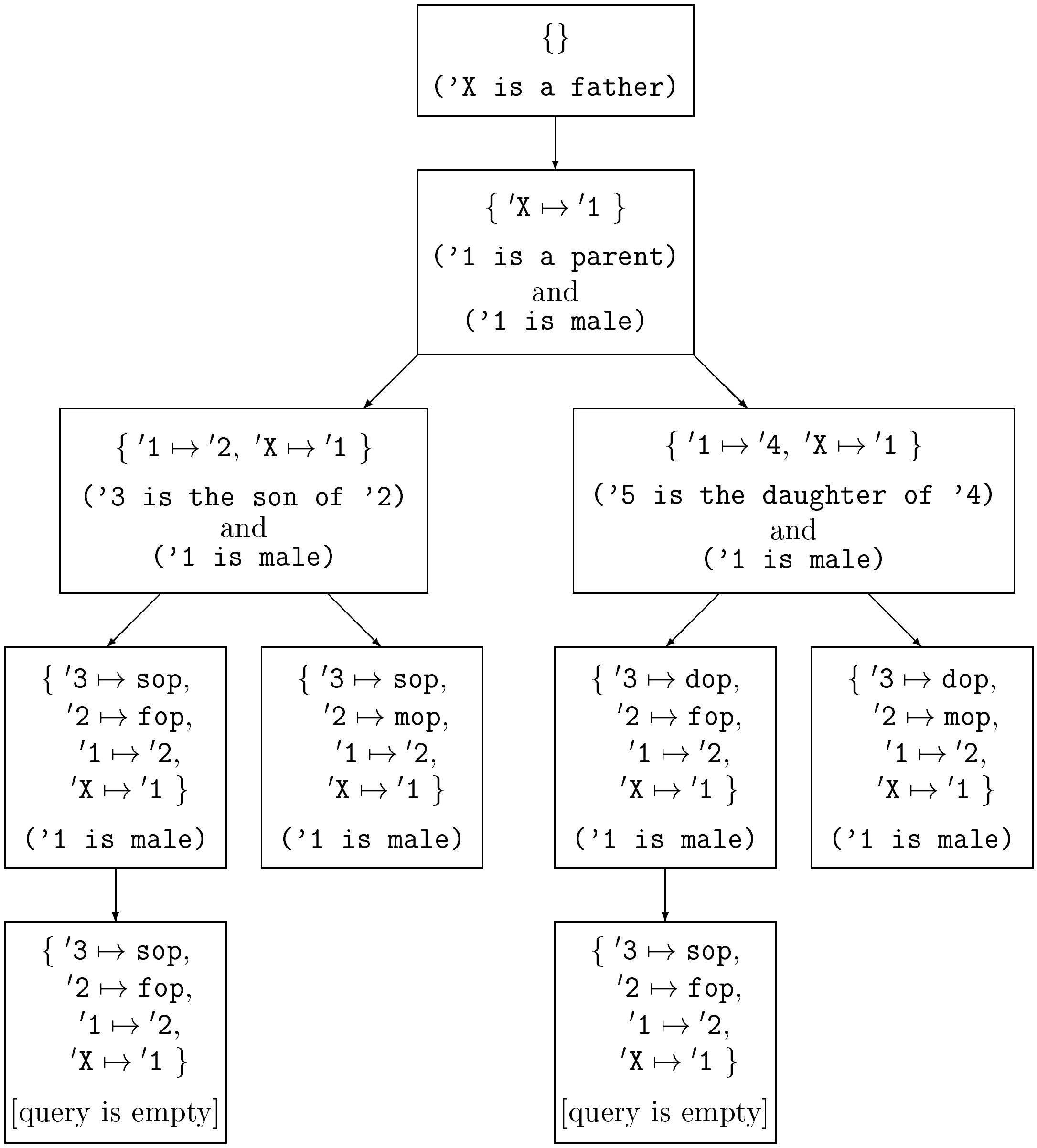


Figure 3.42: SLD-Resolution Search Tree

$\{$ `'1` $\mapsto$ `'4`, `'X` $\mapsto$ `'1` $\}$ `('5 is the daughter of '4) and ('1 is male)`

Continuing, we build the entire search tree, which is displayed in Figure 3.42.

In this figure, the reason that the state:

$\{$ `'3` $\mapsto$ `dop`, `'2` $\mapsto$ `mop`, `'1` $\mapsto$ `'2`, `'X` $\mapsto$ `'1` $\}$ `('1 is male)`

has no children is that when we apply the state's substitution to the state's query we find that `'1`$\mapsto$`'2`$\mapsto$`mop` and so the query is really `(mop is male)` which does not match any definite clause head in the database.

The goals in this search tree are the states with empty query. In general the number of predicates in a query decreases only when the selected predicate matches the head of a definite clause that has no body, and such a match indicates that the selected predicate is true with no pre-conditions.

The search can be depth first or breadth first. Often all goal states need to be found, in which case the search should print out the substitution of each goal state when it is found, and then continue the search. In printing the substitution, one generally prints each non-generated variable mapped by the substitution, and the expression that replaces that variable after substitutions are made recursively until no more substitutions can be made. Thus in the example of Figure 3.42, both goal states have `'X`$\mapsto$`'1`$\mapsto$`'2`$\mapsto$`fop`, so `'X`$\mapsto$`fop` would be printed.

Pseudocode for the SLD-resolution algorithm children function is given in Figure 3.43. The matching of the selected predicate with a clause head is referred to as the unification of two predicates. This is discussed in detail below.

Exercise 3.4.2.1 For the database of exercise 3.4.1 draw a path in the SLD-resolution search tree from the root (inital state) to the goal node with the given answer, for each of the following query/answer pairs. Each node in the path should be carefully labeled as in Figure 3.42.

(a) `(path b to 'Y)?` $\{$ `'Y` $\mapsto$ `e` $\}$
(b) `(path c to 'Y)?` $\{$ `'Y` $\mapsto$ `f` $\}$
(c) `(path d to 'Y)?` $\{$ `'Y` $\mapsto$ `e` $\}$
(d) `(path 'X to d)?` $\{$ `'X` $\mapsto$ `b` $\}$
(e) `(path 'X to e)?` $\{$ `'X` $\mapsto$ `c` $\}$
(f) `(path 'X to f)?` $\{$ `'X` $\mapsto$ `b` $\}$

To find the children of a state:

(1) Initialize the list of children to empty.

(2) Select the first predicate of the state's query. If the query is empty, terminate with an empty list of children.

(3) For each definite clause in the database:

  (a) Replace all the variables in the clause by newly generated variables.

  (b) Unify (match) the selected query predicate with the head of the clause with newly generated variables. Use the substitution of the current state as input to the unification.

  (c) If the unification succeeds, add a new child to the list of children. The substitution of this child is the output of the unification. The query is the original state's query with the selected predicate replaced by the body of the definite clause with newly generated variables.

Figure 3.43: SLD-Resolution Children Function Pseudo-Code

3.4.3 Unification

The **unification algorithm** takes two s-expressions that may contain variables and computes a substitution that makes the two s-expressions syntactically identical. It is convenient for the algorithm to take two s-expressions and a substitution as input, and apply the substitution to the input s-expressions as needed. The output substitution equals the input substitution plus new variable mappings.

Of course the unification algorithm may fail to find any substitution that makes the s-expressions syntactically identical, and therefore the algorithm may fail.

The unification algorithm is implemented by the '`unify`' function defined by rewrite rules in Figure 3.45. The first step is to check the two input expressions to see if they are variables that are mapped by the substitution. If yes, the variable expressions are replaced by the expressions they map to, recursively, until the inputs are unmapped variables.

Then if the two expressions are identical unmapped variables unification succeeds without modifying the input substitution, which is returned as the output substitution. Otherwise, if either input expression is an unmapped variable, a mapping from that variable to the other input is added to the substitution, as this creates a substitution that makes the inputs syntactically identical. The resulting augmented substitution is returned from the function.

Next, if either input is an atom, then if both inputs are the same atom, the unification succeeds returning the input substitution unmodified. But if either input is an atom not `eql` to the other input, unification fails, and the '`unify`' function returns `'fail`.

Lastly, if both inputs are cons cells, the unification algorithm is run first on the `car`'s of the two cons cells. If this fails the total unification fails. Otherwise, the unification algorithm is run second on the `cdr`'s of the two cons cells, using the substitution output by the `car` unification as input to the `cdr` unification, and returning the output of the `cdr` unification as final output (this might or might not be `'fail`).

In the LISP code substitutions are represented by association lists. Thus $\{\ '\mathtt{X} \mapsto '\mathtt{1},\ '\mathtt{Y} \mapsto '\mathtt{2}\ \}$ is represented by `( (X '1) (Y '2) )`. See Figure 3.41 for a precise grammar of substitutions in LISP. This format for substitutions was chosen so that the COMMONLISP '`assoc`' function could be used to lookup a variable name in a substitution.

Code for the '`unify`' function is in Figure 3.46. This code uses two small helper functions, `isvar` and `varname`, defined at the top of Figure 3.44. Some sample output is in Figure 3.47.

Code for the '`subin`' function the performs 'substitutions in' an s-expression, and for the '`variables`' function that returns a list of all the variable names in an s-expression are in Figures 3.48 and 3.49, and example uses are in Figure 3.50. These functions are not needed by '`unify`' itself, but can be useful for using the output of '`unify`'.

Exercise 3.4.3.1 For each of the following sets of inputs, give the output of the '`subin`' function as defined by the rewrite rules in Figure 3.49

(a) `('X + 'Y)` $\{\ '\mathtt{X} \mapsto (5+9)\ \}$
(b) `('X + 'Y)` $\{\ '\mathtt{X} \mapsto ('\mathtt{Y}+9),\ '\mathtt{Y} \mapsto 77\ \}$
(c) `('X + 'Y)` $\{\ '\mathtt{X} \mapsto ('\mathtt{Y}+9),\ '\mathtt{Y} \mapsto (77+'\mathtt{X})\ \}$
(d) `('X + 'Y)` $\{\ '\mathtt{X} \mapsto ('\mathtt{W}+9),\ '\mathtt{W} \mapsto (77+'\mathtt{Y})\ \}$
(e) `('X + 'Y)` $\{\ '\mathtt{X} \mapsto ('\mathtt{W}+9),\ '\mathtt{Y} \mapsto (77+'\mathtt{W})\ \}$

```
;; Helper functions to improve readability:
;;
(defun isvar (e) (and (consp e) (eql (car e) 'quote)))
(defun varname (e) (cadr e))


;; Integer used to generate new variables '1, '2, ... .
;;
(defvar *next-generated-name* 1)

;; Function to return the next generated name (from the
;; list 1, 2, 3, ...).  Just increments *next-generated-
;; name* returning its value before incrementing.
;;
(defun generate-variable-name ()
  (let ((j *next-generated-name*))
       (setf *next-generated-name* (+ j 1))
       j))
```

Figure 3.44: Unification Helper Functions

Exercise 3.4.3.2 For each of the following sets of inputs, give the output of the '`unify`' function as defined by the rewrite rules in Figure 3.45.

(a)	`('X + 'Y)`	`'Z`	$\{\ '\mathtt{Z} \mapsto (5+9)\ \}$
(b)	`('X + 'Y)`	`'Z`	$\{\ '\mathtt{Z} \mapsto ('\mathtt{Y}+9)\ \}$
(c)	`('X + 'X)`	`'Z`	$\{\ '\mathtt{Z} \mapsto (5+9)\ \}$
(d)	`('X + 'Y)`	`'Z`	$\{\ '\mathtt{X} \mapsto (5+9)\ \}$
(e)	`('X + 'Y)`	`('Z + 5)`	$\{\ '\mathtt{X} \mapsto (1*2)\ \}$

```
;; Find the most general unifier (mgu) of two s-expressions
;; (1st and 2nd arguments).  A substitution (3rd argument) to be
;; applied to the s-expressions may also be input (defaults to ()).
;; Returns 'FAIL if the s-expressions do not unify, or the most
;; general unifier if the s-expressions do unify.  The returned
;; substitution is actually the most general unifier prefixed to
;; any input substitution.  This function does no occurs check.
;;
;; The following must be applied in order:
;;
;;      (unify 'x 'y) ===> (unify 'x 'y '())
;;      (unify ''v 'y 's) ===> (unify 'e 'y 's)
;;              if (assoc 'v 's) ===> '(v e)
;;      (unify 'x ''v 's) ===> (unify 'x 'e 's)
;;              if (assoc 'v 's) ===> '(v e)
;;      (unify ''v 'v 's) ===> 's
;;      (unify ''v 'y 's) ===> '((v y) . s)
;;      (unify 'x ''v 's) ===> '((v x) . s)
;;      (unify 'a 'y 's) ===> 's        if (eql 'a 'y)
;;                       ===> 'fail     otherwise
;;      (unify 'x 'a 's) ===> 's        if (eql 'x 'a)
;;                       ===> 'fail     otherwise
;;      (unify '(x1 . y1) '(x2 . y2) 's)
;;          ===> 'fail if (unify 'x1 'x2 's) ===> 'fail
;;          ===> (unify 'y1 'y2 's2)
;;                     if (unify 'x1 'x2 's) ===> 's2
;;
;;      where x, x1, x2, y, y1, y2 are s-expressions;
;;            v is a variable name; a is an atom;
;;            s, s2 are substitutions
```

Figure 3.45: The Unify Function: Page 1 of 2

```
;; Since assoc is expensive, effort is made to avoid
;; calling it twice to do the same thing.
;;
(defun unify (x y &optional (s nil))
  (cond
   ((isvar x)
    (let* ((vx (varname x))
           (rx (assoc vx s)))
          (cond
           (rx (unify (cadr rx) y s))
           ((null rx)
            (cond
             ((isvar y)
              (let* ((vy (varname y))
                     (ry (assoc vy s)))
                    (cond
                     (ry (unify x (cadr ry) s))
                     ((eq vx vy) s)
                     (t `((,vx ,y) . ,s)))))
             (t `((,vx ,y) . ,s)))))))
   ((isvar y)
    (let* ((vy (varname y))
           (ry (assoc vy s)))
          (cond
           ((null ry) `((,vy ,x) . ,s))
           (t (unify x (cadr ry) s)))))
   ((or (atom x) (atom y))
    (cond ((eql x y) s)
          (t 'fail)))
   ((and (consp x) (consp y))
    (let ((s2maybe (unify (car x) (car y) s)))
         (cond
          ((eql s2maybe 'fail) 'fail)
          (t (unify (cdr x) (cdr y) s2maybe)))))))
```

Figure 3.46: The Unify Function: Page 2 of 2

```
(unify '('x is a man) '(bob is a man))
--->
((X BOB))

(unify '('x + 'y) '(5 + 'x))
--->
((Y 5) (X 5))

(unify '('x + 'y) '(5 + 'w))
--->
((Y 'W) (X 5))

(unify '('x + 'y) '(5 + 'w) '((w 'x)))
--->
((Y 5) (X 5) (W 'X))

(unify '('x + 'w) '(5 + 'y) '((w 'x)))
--->
((Y 5) (X 5) (W 'X))

(unify '5 '6)
--->
FAIL

(unify '('x + 'x) '(5 + 6))
--->
FAIL

(unify '('x is a woman) '(bob is a man))
--->
FAIL
```

Figure 3.47: `Unify` Function Test Output

```
;; Returns a list of the names of all the variables in
;; an s-expression (1st and only argument).  Names are
;; not duplicated in the returned list.
;;
;;      (variables 'x) ===> (variables 'x '())
;;      (variables 'a 'vl) ===> 'vl
;;      (variables ''v 'vl)
;;          ===> 'vl        if (member 'v 'vl)
;;          ===> '(v . vl)  otherwise
;;      (variables '(x . y) 'vl) ===>
;;          (variables 'x (variables 'y 'vl))
;;
;;      where x, y are s-expressions; a is an atom;
;;            v is a variable-name;
;;           vl is a list of variable-names;
;;
(defun variables (e &optional (vl nil))
  (cond
   ((atom e) vl)
   ((isvar e)
    (cond
     ((member (varname e) vl) vl)
     (t `(,(varname e) . ,vl))))
   ((consp e)
    (variables (car e) (variables (cdr e) vl)))))
```

Figure 3.48: The `Variables` Function

```
;; Perform variable substitutions on an s-expression.
;; Takes an expression (1st argument) and a substitution
;; (2nd argument) and returns a copy of the expression
;; with substitutions made.  No substitutable variable
;; appears in the output.
;;
;;       (subin 'a 's) ===> 'a
;;       (subin ''v 's)
;;           ===> ''v    if (assoc 'v 's) ===> nil
;;           ===> (subin 'e 's)
;;                       if (assoc 'v 's) ===> '(v e)
;;       (subin '(x . y) 's) ===>
;;               (cons (subin 'x 's) (subin 'y 's))
;;
;;       where a is an atom; s is a substitution;
;;             v is a variable name;
;;             e, x, y are s-expressions
;;
(defun subin (e s)
  (cond
   ((atom e) e)
   ((isvar e)
    (let ((r (assoc (varname e) s)))
         (cond
          ((null r) e)
          (t (subin (cadr r) s)))))
   ((consp e)
    (cons (subin (car e) s) (subin (cdr e) s)))))
```

Figure 3.49: The Substitution Function

```
(subin '(this is 'x 'y) '((x an) (y s-expression)))
--->
(THIS IS AN S-EXPRESSION)

(subin '(('x * 'y) + ('x * 'z)) '((x 5) (z 10)))
--->
((5 * 'Y) + (5 * 10))

(subin '(('x * 'y) + ('x * 'z))
       '((x 'w) (w 'p) (q 10) (p 5) (z 'q)))
--->
((5 * 'Y) + (5 * 10))

(variables '('x + ('y * 'z)))
--->
(X Y Z)

(variables '(('w * 'x) + ('y * 'z)))
--->
(W X Y Z)

(generate-variable-name)
--->
1

(generate-variable-name)
--->
2
```

Figure 3.50: `Subin` and `Variables` Function Test Output

3.4.4 Logic Programming in LISP

To code the SLD-algorithm in LISP, all we need to is program the data abstraction part of a goal search. Code to do this, except for the '`children`' function, appears in Figures 3.51 and 3.52. This code consists of some global variables described in the figures, some helper functions to get the head and body of a definite clause and handle lists of predicates (which have '`and`'s separating the predicates for readability), and a '`goal`' function.

The data structures used by this code are given in Figure 3.41.

The '`goal`' function has two modes of operation. If `*print-substitutions*` is false (`nil`), then the goal function returns true if its argument is a goal state (state with an empty query), and returns false otherwise. If this global variable is true (`t`), then the goal function always returns `nil` to keep the search going, and if its argument is a goal state, the '`goal`' function prints '`SUCCESS`' and then prints out a summary of the state's substitution. The code to print this substitution summary is left as an exercise for the reader: a typical output would be:

```
SUCCESS
'PARENT +--> fop
'PARENT +--> mop
```

Code for the '`children`' function is left as an exercise for the reader. The pseudo-code is given in Figure 3.43.

The database given as an example at the beginning of Section 3.4 is rendered into LISP in Figure 3.53 and queries are input in Figures 3.54 and 3.55. For these tests, the `*print-substitutions*` switch is set to true so on success the substitutions print, but the search always continues to the end and then returns `nil`, and does not return the path to the goal, for the search engine thinks it has failed. The last test is traced by the search engine (`*trace*` is set true).

One of the most interesting uses of logic programming is for parsing **natural language** (e.g., English). A database, i.e., logical program, to parse a very small subset of of English is given in Figures 3.56, 3.57, and 3.58. Tests are given in Figures 3.59 and 3.60.

Some predicates from this example are:

```
((sentence from 'P1 to 'P3)
 <=== (noun phrase from 'P1 to 'P2 with number 'N)
      and
      (verb phrase from 'P2 to 'P3 with number 'N))
```

and

```
((word 'WORD from ('WORD . 'P2) to 'P2))
```

The variables `'P1`, `'P2`, and `'P3` denote positions between words of a phrase. The phrase is input as a LISP list, the input phrases in the tests being:

```
(shrdlu concerns bill)
(shrdlu concern bill)
(the professor wrote the book that concerns lunar)
(the professor wrote the book that concern lunar)
```

Two of these phrases parse, and to do not because of 'number disagreement'. Here '**number**' is either '**singular**' or '**plural**'.

A position in a phrase is denoted by the remainder of the phrase after the position. Thus the positions in the first phrase above are:

```
(shrdlu concerns bill)
(concerns bill)
(bill)
()
```

This method of representing positions has been chosen so that with the help of the single short predicate:

```
((word 'WORD from ('WORD . 'P2) to 'P2))
```

we can deduce that the appropriate words are between the appropriate positions. Thus for the four positions just given we can deduce:

```
(word shrdlu from (shrdlu concerns bill) to (concerns bill))
(word concerns from (concerns bill) to (bill))
(word bill from (bill) to ())
```

The first test of the natural language example has `*print-substitutions*` set false so that the search finds a single goal and the search engine prints the path to that goal. The end of the path, which is printed first, is a state with an empty query whose substitution includes all the other substitutions on the path. Therefore the substitutions for all the other states have been omitted to save space. The queries for all states are given, though for the first state printed the query is empty.

Logic programming is the basis of the programming language PROLOG. For more information about PROLOG and the use of logic programming to parse natural language, see [PS87].

Exercise 3.4.4.1

(a) Write the rewrite rules for the '`children`' function for logic programming. The pseudo-code is given in Figure 3.43.

(b) Write the LISP code for this function.

(c) Add the LISP code to print substitutions to the '`goal`' function of Figure 3.52.

```
;; Database (list of definite clauses):
;;
(defvar *database* nil)

;; Switch that is true to print substitutions for all
;; goals but not to print the path to the first goal.
;; This is done by having GOAL print the 'SUCCESS'
;; and the substitution for each goal state, while
;; returning NIL always to keep the search going.
;;
;; False to print the path to first goal found and stop
;; search at the first goal.  This is done by having
;; GOAL do its normal thing of returning true for
;; goal states, so the search engine stops and returns
;; the path which prints as a return value.
;;
(defvar *print-substitutions* t)


;; Helper functions to make code more readable.
;;
(defun head (definite-clause) (car definite-clause))
(defun body (definite-clause) (cddr definite-clause))
(defun predicate-car (predicate-list)
  (car predicate-list))
(defun predicate-cdr (predicate-list)
  (cddr predicate-list))
(defun predicate-cons (predicate predicate-list)
  (cond ((null predicate-list) `(,predicate))
        (t `(,predicate and . ,predicate-list))))
(defun predicate-append
          (predicate-list1 predicate-list2)
  (cond
   ((null predicate-list1) predicate-list2)
   (t
    (predicate-cons (predicate-car predicate-list1)
                    (predicate-append
                     (predicate-cdr predicate-list1)
                     predicate-list2)))))
```

Figure 3.51: SLD-Resolution Algorithm Code: Page 1 of 2

```
;; Function to generate a substitution that will
;; replace all the variables in an s-expression by
;; newly generated variables.
;;
;; Generated names are created by GENERATE-
;; VARIABLE-NAME.
;;
(defun generate-variables (e)
  (mapcar #'(lambda (v)
                    `(,v ',(generate-variable-name)))
          (variables e)))


;; GOAL function.  The simplest GOAL function just
;; returns true if and only if the state has an empty
;; predicate list.  The more complicated goal functions
;; print `SUCCESS' and the state substitution when the
;; state has an empty predicate list, but always return
;; false to keep the search going.
;;
(defun goal (state)
  (cond
   ((not *print-substitutions*)
    (null (cdr state)))
   ((null (cdr state))
    ;; Print `SUCCESS' and state substitution.

    (fresh-line)
    (princ '|SUCCESS|)
    (terpri)

    ;; Substitution print code goes here.

    (terpri)

    ;; Return NIL to keep search going.
    nil)))
```

Figure 3.52: SLD-Resolution Algorithm Code: Page 2 of 2

```
;; Begin tests of SLD-Resolution search.

;; Sets database and returns NIL to prevent large
;; printout.
;;
(defun set-database (value)
  (setf *database* value)
  nil)


;; Ancestry Example

(set-database
  '(    ((fop is male))
        ((mop is female))
        ((dop is female))
        ((sop is male))
        ((sop is the son of fop))
        ((sop is the son of mop))
        ((dop is the daughter of fop))
        ((dop is the daughter of mop))

        (('X is a parent) <=== ('Y is the son of 'X))
        (('X is a parent) <=== ('Y is the daughter of 'X))
        (('X is a father) <=== ('X is a parent)
                               and ('X is male))))
```

Figure 3.53: Ancestry Example in LISP

```
(breadth-first-search '((() . (('X is female))))
  #'children #'goal)
--->
SUCCESS
'X +--> DOP

SUCCESS
'X +--> MOP

NIL

(depth-first-search '((() . (('X is female))))
  #'children #'goal)
--->
SUCCESS
'X +--> DOP

SUCCESS
'X +--> MOP

NIL

(breadth-first-search '((() . (('P is a parent))))
  #'children #'goal)
--->
SUCCESS
'P +--> MOP

SUCCESS
'P +--> FOP

SUCCESS
'P +--> MOP

SUCCESS
'P +--> FOP

NIL
```

Figure 3.54: Test of Ancestry Example in LISP: Page 1 of 2

```
;; *trace* is set to t
;;
(breadth-first-search '((() . (('F is a father))))
  #'children #'goal)
--->
Processing State: (NIL ('F IS A FATHER))
New States: ((((F '36)) ('36 IS A PARENT) AND ('36 IS MALE)))
Processing State: (((F '36)) ('36 IS A PARENT) AND ('36 IS MALE))
New States:
    ((((36 '40) (F '36)) ('39 IS THE DAUGHTER OF '40) AND ('36 IS MALE))
     (((36 '38) (F '36)) ('37 IS THE SON OF '38) AND ('36 IS MALE)))
Processing State:
(((36 '40) (F '36)) ('39 IS THE DAUGHTER OF '40) AND ('36 IS MALE))
New States: ((((40 MOP) (39 DOP) (36 '40) (F '36)) ('36 IS MALE))
             (((40 FOP) (39 DOP) (36 '40) (F '36)) ('36 IS MALE)))
Processing State:
(((36 '38) (F '36)) ('37 IS THE SON OF '38) AND ('36 IS MALE))
New States: ((((38 MOP) (37 SOP) (36 '38) (F '36)) ('36 IS MALE))
             (((38 FOP) (37 SOP) (36 '38) (F '36)) ('36 IS MALE)))
Processing State: (((40 MOP) (39 DOP) (36 '40) (F '36)) ('36 IS MALE))
New States: NIL
Processing State: (((40 FOP) (39 DOP) (36 '40) (F '36)) ('36 IS MALE))
New States: ((((40 FOP) (39 DOP) (36 '40) (F '36))))
Processing State: (((38 MOP) (37 SOP) (36 '38) (F '36)) ('36 IS MALE))
New States: NIL
Processing State: (((38 FOP) (37 SOP) (36 '38) (F '36)) ('36 IS MALE))
New States: ((((38 FOP) (37 SOP) (36 '38) (F '36))))
Processing State: (((40 FOP) (39 DOP) (36 '40) (F '36)))
SUCCESS
'F +--> FOP

New States: NIL
Processing State: (((38 FOP) (37 SOP) (36 '38) (F '36)))
SUCCESS
'F +--> FOP

New States: NIL
NIL
```

Figure 3.55: Test of Ancestry Example in LISP: Page 2 of 2

```
;; Natural Language Parsing Example

(set-database
  '(    ((sentence from 'P1 to 'P3)
         <=== (noun phrase from 'P1 to 'P2 with number 'N)
              and
              (verb phrase from 'P2 to 'P3 with number 'N))

        ((noun phrase from 'P1 to 'P4 with number 'N)
         <=== (determiner from 'P1 to 'P2 with number 'N)
              and
              (noun from 'P2 to 'P3 with number 'N)
              and
              (optional relative clause from 'P3 to 'P4 with number 'N))

        ((noun phrase from 'P1 to 'P2 with number 'N)
         <=== (proper noun from 'P1 to 'P2 with number 'N))

        ((verb phrase from 'P1 to 'P3 with number 'N)
         <=== (transitive verb from 'P1 to 'P2 with number 'N)
              and
              (noun phrase from 'P2 to 'P3 with number 'ANY))

        ((verb phrase from 'P1 to 'P2 with number 'N)
         <=== (intransitive verb from 'P1 to 'P2 with number 'N))

        ((optional relative clause from 'P1 to 'P1
                   with number 'ANY))

        ((optional relative clause from 'P1 to 'P3
                   with number 'N)
         <=== (word that from 'P1 to 'P2)
              and
              (verb phrase from 'P2 to 'P3 with number 'N))
```

Figure 3.56: Natural Language Parsing Example: Page 1 of 3

```
((determiner from 'P1 to 'P2 with number 'N)
 <=== (word 'WORD from 'P1 to 'P2)
      and
      ('N determiner 'WORD))

((singular determiner a))
((singular determiner every))
(('ANY determiner the))
((plural determiner some))

((noun from 'P1 to 'P2 with number 'N)
 <=== (word 'WORD from 'P1 to 'P2)
      and
      ('N noun 'WORD))

((singular noun author))
((plural   noun authors))
((singular noun book))
((plural   noun books))
((singular noun professor))
((plural   noun professors))
((singular noun program))
((plural   noun programs))
((singular noun student))
((plural   noun students))

((proper noun from 'P1 to 'P2 with number 'N)
 <=== (word 'WORD from 'P1 to 'P2)
      and
      ('N proper noun 'WORD))

((singular proper noun bill))
((singular proper noun bertrand))
((singular proper noun gottlob))
((singular proper noun terry))
((singular proper noun begriffsschrift))
((singular proper noun principia))
((singular proper noun lunar))
((singular proper noun shrdlu))
```

Figure 3.57: Natural Language Parsing Example: Page 2 of 3

```
        ((transitive verb from 'P1 to 'P2
                     with number 'N)
         <=== (word 'WORD from 'P1 to 'P2)
              and
              ('N transitive verb 'WORD))

        ((singular transitive verb concerns))
        ((plural transitive verb concern))
        (('ANY transitive verb met))
        (('ANY transitive verb ran))
        (('ANY transitive verb wrote))

        ((intransitive verb from 'P1 to 'P2
                       with number 'N)
         <=== (word 'WORD from 'P1 to 'P2)
              and
              ('N intransitive verb 'WORD))

        ((singular intransitive verb halts))
        ((plural intransitive verb halt))

        ((word 'WORD from ('WORD . 'P2) to 'P2))

        ))


(defun test-sentence (phrase)
  (breadth-first-search
   `((() . ((sentence from ,phrase to ()))))
   #'children #'goal))
```

Figure 3.58: Natural Language Parsing Example: Page 3 of 3

```
;; *print-substitutions* is NIL to show the path.

(test-sentence '(shrdlu concerns bill))
--->
((((889 SINGULAR) (1007 NIL) (1006 BILL) (890 '1006)
      (811 '889) (888 NIL) (887 (BILL)) (504 '811)
      (810 NIL) (809 (BILL)) (632 (BILL)) (747 (BILL))
      (746 CONCERNS) (634 '746) (633 SINGULAR)
      (502 '632) (631 (CONCERNS BILL)) (501 SINGULAR)
      (503 NIL) (500 (CONCERNS BILL)) (213 SINGULAR)
      (212 (CONCERNS BILL)) (331 (CONCERNS BILL))
      (330 SHRDLU) (214 '330) (135 '213) (134 '212)
      (211 (SHRDLU CONCERNS BILL)) (75 '135) (73 '134)
      (133 (SHRDLU CONCERNS BILL)) (74 NIL)
      (72 (SHRDLU CONCERNS BILL))))  ;; Here query is NIL
  ( ...omitted substitution...
    ('889 PROPER NOUN '890))
  ( ...omitted substitution...
    (WORD '890 FROM '887 TO '888) AND
    ('889 PROPER NOUN '890))
  ( ...omitted substitution...
    (PROPER NOUN FROM '809 TO '810 WITH NUMBER '811))
  ( ...omitted substitution...
    (NOUN PHRASE FROM '502 TO '503 WITH NUMBER '504))
  ( ...omitted substitution...
    ('633 TRANSITIVE VERB '634) AND
    (NOUN PHRASE FROM '502 TO '503 WITH NUMBER '504))
  ( ...omitted substitution...
    (WORD '634 FROM '631 TO '632) AND
    ('633 TRANSITIVE VERB '634) AND
    (NOUN PHRASE FROM '502 TO '503 WITH NUMBER '504))
  ( ...omitted substitution...
    (TRANSITIVE VERB FROM '500 TO '502 WITH NUMBER '501)
    AND (NOUN PHRASE FROM '502 TO '503 WITH NUMBER '504))
```

Figure 3.59: Natural Language Parsing Test: Page 1 of 2

```
  ( ...omitted substitution...
    (VERB PHRASE FROM '73 TO '74 WITH NUMBER '75))
  ( ...omitted substitution...
    ('213 PROPER NOUN '214) AND
    (VERB PHRASE FROM '73 TO '74 WITH NUMBER '75))
  ( ...omitted substitution...
    (WORD '214 FROM '211 TO '212) AND
    ('213 PROPER NOUN '214) AND
    (VERB PHRASE FROM '73 TO '74 WITH NUMBER '75))
  ( ...omitted substitution...
    (PROPER NOUN FROM '133 TO '134 WITH NUMBER '135) AND
    (VERB PHRASE FROM '73 TO '74 WITH NUMBER '75))
  ( ...omitted substitution...
    (NOUN PHRASE FROM '72 TO '73 WITH NUMBER '75) AND
    (VERB PHRASE FROM '73 TO '74 WITH NUMBER '75))
  (NIL (SENTENCE FROM (SHRDLU CONCERNS BILL) TO NIL))
)

;; Do not show paths to keep printout from being
;; extremely large.

(setf *print-substitutions* t)
--->
T

(test-sentence '(shrdlu concern bill))
--->
NIL

(test-sentence '(the professor wrote the book that concerns lunar))
--->
SUCCESS

NIL

(test-sentence '(the professor wrote the book that concern lunar))
--->
NIL
```

Figure 3.60: Natural Language Parsing Test: Page 2 of 2

Bibliography

[Gra96] Paul Graham. *ANSI Common Lisp.* Prentice-Hall, 1996.

[LP81] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, 1981.

[Mit96] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[PS87] Fernando C. N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis.* CSLI Publications, 1987.

[Rob84] Fred S. Roberts. *Applied Combinatorics.* Prentice-Hall, 1984.