Logiweb

Klaus Grue

January 3, 2003

Contents

1 Introduction

Logiweb is a system that resembles the World Wide Web (WWW) in that it allows people to make information accessible to other people. Logiweb differs from WWW in that it is specially tuned towards presentation of mathematics and computer science. Mathematics includes presentation of mathematical typography, axiomatic systems, lemmas, and formal proofs in such a way that computers can check the proofs. Computer science includes presentation of programming languages, programs, design and requirement documents and test reports in such a way that computers can execute the programs and can make soundness checks.

1.1 Components

The Logiweb system has a number of components:

Web The Logiweb web is a varying collection of Logiweb pages. Each page consists of a bit vector and a bibliography where the bibliography is a list of pointers to other Logiweb pages. The pages and bibliography pointers form the nodes and edges, respectively, of a directed, acyclic graph.

Server Logiweb servers are programs that cooperate on maintaining a web. Users may submit pages to servers which then make the pages available to other users.

Browser Logiweb browsers are programs that allow a user to view and manipulate pages.

Codex Every Logiweb page contains a set of definitions. Such a set of definitions is called a Logiweb codex. Logiweb specifies an algorithm which, given a page in a web, returns the codex of the page. The codex of a page may contain definitions of computable functions, axioms, axiom systems, lemmas, and proofs. The codex of a page may also define how to display the page on a screen, how to print the page, and what interactive behaviour the page has. The codex may also contain definitions of letter shapes and font metric data. In general, a codex can contain any definition that can be formalised and stored on a computer.

Abstract machine The Logiweb machine is an abstract machine that can execute functional programs and can interact with the world around it. The Logiweb machine is a Turing complete computing machine. When a Logiweb machine is started, it bootstraps from a Logiweb page. What the machine does after that depends of the contents of that boot page. The Logiweb browser is intended to run on a Logiweb machine.

Logiweb gates A Logiweb gate is a http reference to a Logiweb page.

Logiweb citation index The Logiweb citation index allows users to establish associations between pages. The pages and the citation index form a general, colored graph as opposed to the bibliographic references that form an acyclic graph.

User interface The Logiweb user interface is the interface of the Logiweb browser.

1.2 Open questions

How should other data types be embedded in S-expressions? Possible solutions:

- Quote solution. This involves modification of eval and puts substitution of equals to danger.
- Constructor solution. Easy. One may consider Quote as a macro that expands $(x :: y)$ to $\langle \mathbf{A}, \langle \mathbf{A}, \langle \mathbf{P} \rangle, x \rangle, y \rangle$ (or $\langle \mathbf{P}, x, y \rangle$).
- Keyword package solution. Requires special handling af those packages that do have proper atoms. Gives tags for free.
- Extra ID solution. Costs some extra memory. Gives tags for free. Allows one page to contain several litteral types.

ANSWER: The "Symbols" section has been rewritten such that a symbol now has form $\langle p, i_1, \ldots, i_q \rangle$. This is essentially the "extra ID" solution.

SECOND THOUGHT: No. It complicates the value function. The constructor solution is clean. Use it.

Should $\langle \mathrm{Interrupt}, \mathrm{Exit}, p \rangle$ be changed to $\langle \mathrm{Interrupt}, \langle \mathrm{Exit} \rangle, p \rangle$? Benefit: Fixed position of p combined with flexible number of parameters of interrupts. YES. And the $\langle \mathrm{Interrupt}, \ldots, p \rangle$ layout should be kept so that p does not need to be included in each and every interrupt term.

Should input events, output events, and interrupt identifiers be described in the representation chapter?

Should Logiweb pages have an unpack function? YES.

Should the "data structures" and "representation" chapters be combined?

Should boot/value representation be de-Curried? Could allow to apply **S**, **K**, and **P** to parameters without the use of **A**. In that case **A** should be replaced by CI. NO. The basic constructs should follow the standard. That allows to display basic constructs without special attention.

2 The environment of Logiweb

2.1 Hardware platform

At the hardware level, Logiweb is supported by a number of *host machines* that are interconnected by a number of *host networks*. As an example, if Logiweb runs on a number of pc's that are interconnected by the internet, then the pc's are the host machines and the internet is the host network.

Computers and networks that host Logiweb need not be dedicated to Logiweb alone. Rather, computers and networks that host Logiweb will typically host other systems as well.

2.2 Logiweb nodes and messages

At any time, Logiweb comprises a number of *Logiweb nodes* and a number of *Logiweb messages*. Logiweb nodes are programs that run on the host machines and which supply the services offered by the Logiweb system. Logiweb messages are messages that the Logiweb nodes send to one another across the host networks.

Examples of Logiweb nodes are Logiweb browsers which allow users to interact with Logiweb. A Logiweb host may host more than one Logiweb node. As an example, two users of a Logiweb host may run a Logiweb browser each.

Logiweb messages are messages that are internal to Logiweb and which conform to a specific format. Logiweb nodes are able to send a receive such messages. Logiweb nodes may also be able to send and receive messages that are not Logiweb messages. As an example, a Logiweb browser that uses the X11 window system will use X11 messages to communicate with the X11 server. Such X11 messages are not Logiweb messages.

2.3 Software platform

At the software level, Logiweb is supported by a number of *host operating systems*, a number of *host programming languages* and a number of *host protocols*.

A host operating system is a piece of basic software that runs on a host machine and which allows Logiweb nodes to start up on that machine. The host operating system may also supply services needed by the Logiweb nodes after start up. Examples of host operating systems could be Windows or Linux or a PC-BIOS. As an example, one may implement a Logiweb node as an EXE file on a Windows system. In that case, the node may be started up by double clicking the EXE file. As another example, one may implement a Logiweb node as a boot floppy disk and insert the boot floppy in a PC. In that case, the node may be started up by powering up the PC.

Logiweb has its own, native programming language, but some of Logiweb has to be implemented in programming languages different from the native one. As an example, one may use C as host programming language for those parts of Logiweb that cannot be expressed in the native language. As another example, hard core programmers may write e.g. a boot floppy or an EXE file using a hex editor, in which case the machine language of the host machine is used as host programming language.

When Logiweb nodes exchange Logiweb messages across a host network, the messages will be routed and delivered using some protocol such as TCP or UDP. Protocols that are used for transmission of Logiweb messages will be refered to as host protocols.

As mentioned before, Logiweb messages must conform to a specific format. That format and the rules for exchange of such messages will be referred to as the *Logiweb protocol*. A Logiweb message that is transfered using UDP must conform to both the UDP protocol and the Logiweb protocol. Such a message will be said to be a Logiweb/UDP message.

From the point of view of UDP, a Logiweb/UDP message is a UDP message whose cargo happens to conform to the Logiweb protocol. From the point of view of Logiweb, a Logiweb/UDP message is a Logiweb message with some header and trailer bytes which are irrelevant to Logiweb. If, in turn, the UDP message is transfered using IP, i.e. as a UDP/IP message, then the resulting message is a Logiweb/UDP/IP message which contains information that adresses three different systems.

3 Logiweb data structures

3.1 Logiweb binary trees

The most fundamental data structure in the Logiweb system is the *Logiweb binary tree*. Logiweb binary trees serve the same purpose in Logiweb as the byte serves in a traditional computer: It is an efficient and general data structure that is useful when implementing other data structures.

The easiest way to introduce Logiweb binary trees is to introduce a syntax for expressing such trees. Let $\mathcal{B}$ be the syntax class defined by the following BNF definition:

$$\mathcal{B} ::= \mathsf{T} | (\mathcal{B}, \mathcal{B})$$

Some examples of Logiweb binary trees are:

$$\begin{array}{l} \mathsf{T} \\ (\mathsf{T}, \mathsf{T}) \\ ((\mathsf{T}, \mathsf{T}), \mathsf{T}) \\ (\mathsf{T}, (\mathsf{T}, \mathsf{T})) \\ ((\mathsf{T}, \mathsf{T}), (\mathsf{T}, \mathsf{T})) \end{array}$$

The syntax class $\mathcal{B}$ has the property that there is a one-to-one correspondence between Logiweb binary trees and elements of $\mathcal{B}$. In other words, any Logiweb binary tree can be expressed in one and only one way using the syntax above.

From the example above it is obvious that there is a countable infinity of Logiweb binary trees.

3.2 The naive representation

Inside a Logiweb node, a simple (but wasteful) way to represent a Logiweb binary tree is to represent T as a nil pointer and (x, y) as a pointer to a two element record structure. In C, the declaration of the data type `bin` and the nil pointer `T` could read:

```
typedef struct bin_record {struct bin_record *head,*tail;} *bin;
#define T ((bin)0)
```

The representation above is too naive (i.e. wasteful w.r.t. memory usage) for most applications, but nevertheless illustrates how Logiweb binary trees could be implemented.

3.3 The Polish prefix representation

When transmitting a Logiweb binary tree over a network, it is important to keep the size of the representation small in order to save band width. The *Polish prefix representation* given in the following is designed for situations where size is a primary concern.

To introduce the Polish prefix representation, we first introduce an alternative syntax $\hat{\mathcal{B}}$ for Logiweb binary trees in which (x, y) is written $\mathsf{P}xy$:

$$\hat{\mathcal{B}} ::= \mathsf{T} | \mathsf{P} \hat{\mathcal{B}} \hat{\mathcal{B}}$$

The "P" in $\mathsf{P}xy$ stands for *pair*.

The Polish prefix representation of a Logiweb binary tree may be obtained thus: Express the tree in the syntax $\hat{\mathcal{B}}$ and then replace T and P by 0 and 1, respectively. The following examples illustrate the translation.

$\mathcal{B}$	$\hat{\mathcal{B}}$	Polish prefix representation
T	T	0
(T, T)	PTT	100
$((\mathsf{T}, \mathsf{T}), \mathsf{T})$	PPTTT	11000
$(\mathsf{T}, (\mathsf{T}, \mathsf{T}))$	PTPTT	10100
$((\mathsf{T}, \mathsf{T}), (\mathsf{T}, \mathsf{T}))$	$\mathsf{PPTTPTT}$	1100100

Hence, as an example, the Polish prefix representation of $(\mathsf{T}, (\mathsf{T}, \mathsf{T}))$ reads 10100. In general, the Polish prefix representation of any Logiweb binary tree is a bit vector that consists of an odd number of bits.

The Polish prefix representation of a Logiweb binary tree always belong to the syntax class *bintree*, which is defined thus:

$$\mathit{bintree} ::= 0 \mid 1 \; \mathit{bintree} \; \mathit{bintree}$$

The Polish prefix representation is *self-terminating* in the sense that if one receives a Logiweb binary tree followed by arbitrary dribble bits, then it is possible to see where the binary tree ends and the dribble begins. As an example, the eight bit pattern 10100110 represents the Logiweb binary tree $(\mathsf{T}, (\mathsf{T}, \mathsf{T}))$ followed by the dribble bits 110.

The Polish prefix representation of a Logiweb binary tree X is space efficient compared to the naive representation except when X contains many repetitions. As an example where the naive representation is superior, define

$$\begin{array}{lll} t_1 & \doteq & \mathsf{T} \\ t_{n+1} & \doteq & (t_n, t_n) \quad (t = 2, 3, \ldots) \end{array}$$

The Polish prefix representation of t_n takes up $2^n - 1$ bits whereas the smallest naive representation of t_n takes up $r(n-1)+p$ bits where r is the size in bits of one record and p is the size in bits of one pointer. For large n, the naive representation is smaller than the Polish prefix one.

The Polish prefix representation is useful for transmission and storage of Logiweb binary trees that contain few repetitions.

3.4 Representation of physical quantities

Logiweb is fundamentalistically metric. Any length is measured in meters. Any time interval is measured in seconds. Any mass is measured in kilograms.

As an example, a font size is a length and, hence, it is measured in meters. It is of course inconvenient for a user to specify a font size of e.g. 0.003 meters, but that is a problem for the designer of user interfaces, not something that concerns the core of Logiweb.

As another example, Logiweb needs time stamps. Logiweb time stamps indicate the number of seconds that have elapsed since the zero of Unix time. In other words, Logiweb uses the same clock as Unix. A Logiweb time stamp is a *decimal fraction*. A number d is a decimal fraction if there exist integers m and e such that

$$d = m \cdot 10^e$$

Logiweb also uses derived units. As an example, area is measured in square meters and paper quality in kilograms per square meter.

3.5 Decadic suffixes

For convenience, the following decadic suffixes are used in Logiweb:

Y	yota	10^{24}
Z	zeta	10^{21}
P	peta	10^{18}
E	exa	10^{15}
T	terra	10^{12}
G	giga	10^{9}
M	mega	10^{6}
k	kilo	10^{3}
U	unit	10^{0}
m	mili	10^{-3}
μ	micro	10^{-6}
u	micro	10^{-6}
n	nano	10^{-9}
p	pico	10^{-12}
f	femto	10^{-15}
a	atto	10^{-18}
z	zepto	10^{-21}
y	yocto	10^{-24}

Note that there are two non-standard decadic suffixes: U stands for 10^0 so that a “unit meter” is the same as a meter. Furthermore, u is introduced as an alternative for μ. Also note that only powers of 1000 are included so that centi, deci, deca, and hekto are omitted.

Since Logiweb is a mathematical, logical, and computational system, the decadic suffixes will be attached to numbers rather than physical units. As an example, 3m stands for the number “three mili”, which is the number 0.003. In Logiweb, decadic suffixes extend the syntax for writing numbers rather than extending the syntax for writing physical units.

As an example of use, if a font has size 3m or 3000μ, then the font has size 0.003 meters. Hence, in Logiweb, there is nothing called a “milimeter”, but it is possible to say that a font has size “three-mili meters”.

Use of more than one decadic suffix is legal, so that 3μm denotes “three micro mili”, i.e. $3 \cdot 10^{-9}$. As an example, a mass of 3μm denotes “three micro mili kilograms”, i.e. three micrograms.

Powers of suffixes are also allowed, such that $3\mathtt{m}^2$ denotes “three square mili”, i.e. $3 \cdot 10^{-6}$. As an example, an area of $3m^2$ denotes “three square mili square meters”, i.e. three square milimeter.

In physics, decadic prefixes are called prefixes because they are put in front of physical units. In the Logiweb system, decadic suffixes are called suffixes because they are put after the digits of a number.

3.6 Binary coded decimal representation

In some situations where the size of the representation is important, Logiweb uses the following *binary coded decimal representation* for decimal fractions:

$$
\begin{array}{lcl}
\mathit{digit} & ::= & 0000 \mid 0001 \mid 0010 \mid 0011 \mid 0100 \mid \\
 & & 0101 \mid 0110 \mid 0111 \mid 1000 \mid 1001 \\
\mathit{natural} & ::= & \mathit{digit}^{+} \\
\mathit{plus} & ::= & 1010 \\
\mathit{minus} & ::= & 1011 \\
\mathit{sign} & ::= & \mathit{plus} \mid \mathit{minus} \\
\mathit{integer} & ::= & \mathit{natural} \mid \mathit{sign}\ \mathit{natural} \\
\mathit{fraction} & ::= & \mathit{integer}\ 1111 \mid \mathit{mantisa}\ \mathit{exponent}\ 1111 \\
\mathit{mantisa} & ::= & \mathit{integer} \\
\mathit{exponent} & ::= & \mathit{sign}\ \mathit{natural}
\end{array}
$$

As an example, 3.14 equals $314 \cdot 10^{-3}$, so 3.14 may be represented thus:

    0011 0001 0100 1011 0011 1111

3.14 also equals $3140 \cdot 10^{-4}$, so 3.14 may also be represented thus:

    0011 0001 0100 0000 1011 0100 1111

One may also prepend a plus sign:

    1010 0011 0001 0100 0000 1011 0100 1111

The sign of the mantisa is optional whereas the sign of the exponent is mandatory. The latter is mandatory because the sign of the exponent marks where the mantisa ends and the exponent begins.

$2k$ may be represented as $2 \cdot 10^3$:

    0010 1010 0011 1111

$2k$ may also be represented as 2000:

    0010 0000 0000 0000 1111

The definition *natural* ::= digit^{+} states that a natural number is a sequence of one or more digits. In particular, the empty string does not represent a natural number. Leading zeros are allowed.

Every decimal fraction ends with the pattern 1111. This allows to see where the decimal fraction ends if it occurs inside some other structure.

3.7 The Logiweb-160 global hash function

The Logiweb system uses a global hash function which we shall refer to as *Logiweb-160*. Logiweb-160 is a slightly modified version of *RIPEMD-160*.

Both Logiweb-160 and RIPEMD-160 take a bit-vector v of arbitrary length as input and produces a 160-bit bit-vector as output. Logiweb-160 and RIPEMD-160 coincide on byte vectors. Furthermore, for every collision one can find for Logiweb-160, one can immediately construct a collision for RIPEMD-160 and vice versa.

Logiweb-160 and RIPEMD-160 disagree on the numbering of bits inside bytes in that Logiweb-160 considers the least significant bit to be the first one and RIPEMD-160 considers the most significant bit to be first.

When applied to a sequence of bytes, Logiweb-160 and RIPEMD-160 produce the same result because the different conventions have no effect on the actual computations.

When applied to a sequence v of bits, Logiweb-160 and RIPEMD-160 pack the bits into bytes in different ways and then apply the same algorithm. For this reason, any Logiweb-160 collision can be converted easily into a RIPEMD-160 collision and vice versa.

When applied to a bit vector, the Logiweb-160 algorithm is as follows:

1. Concatenate a "one"-bit at the end of the bit vector.
2. Pad the bit vector with "zero"-bits so that its length is a multiple of eight bits. This adds from zero to seven bits to the vector.
3. Reverse the order of the last eight bits. This reversal ensures that Logiweb-160 and RIPEMD-160 will produce the same hash code when applied to byte vectors.
4. Add "zero"-bits until the bit vector is 64 bits short of being a multiple of 512 bits.
5. Express the length of the original bit vector as a 64 bit binary number with the least significant bit first and concatenate the number at the end of the bit vector. If the original bit vector contains 2^{64} bits or more, use the length modulo 2^{64}.
6. Convert the bit vector into a byte vector by placing bit n of the vector in bit $(n \bmod 8)$ of byte $(n \operatorname{div} 8)$. In each byte, bit 0 and 7 are the least and most significant bits, respectively.
7. Apply RIPEMD-160 (without any padding) to the byte vector.

An alternative (and less efficient) formulation of Logiweb-160 is as follows: Convert the bit vector into a byte vector and pad it to a multiple of 512 bits as specified by RIPEMD-160. Then reverse the order of bits in each byte up to but not including the byte that contains the stop bit.

3.8 Logiweb pages

Logiweb pages constitute another fundamental data structure in Logiweb. The format $\mathcal{P}$ of a Logiweb page reads:

$$\begin{array}{lll}
\mathcal{P} & ::= & \textit{reference contents} \\
\textit{contents} & ::= & \textit{bibliography body} \\
\textit{reference} & ::= & \textit{version key timestamp} \\
\textit{version} & ::= & 0000\ 0001 \\
\textit{key} & ::= & \textit{bit}^{160} \\
\textit{bit} & ::= & 0 \mid 1 \\
\textit{timestamp} & ::= & \textit{fraction} \\
\textit{bibliography} & ::= & \textit{reference}^{*}\ 0000\ 0000 \\
\textit{body} & ::= & \textit{bintree}
\end{array}$$

The definition $\textit{key} ::= \textit{bit}^{160}$ says that a *key* consists of exactly 160 bit.

As can be seen from the syntax, every Logiweb page is a bit vector.

Every Logiweb page consists of a *version* followed by a *key* followed by a sequence of bits. The sequence of bits that follow the *key* will be referred to as the *vector* of the page. Hence, an alternative definition of the syntax class $\mathcal{P}$ of Logiweb pages read:

$$\begin{array}{lll}
\mathcal{P} & ::= & \textit{version key vector} \\
\textit{vector} & ::= & \textit{timestamp contents}
\end{array}$$

The *version*, the *key*, and the *vector* of a Logiweb page are the three bit vectors that arise when splitting up the page according to the syntax

$$\mathcal{P} ::= \textit{version key vector}$$

The *timestamp* and the *contents* of a Logiweb page are the three bit vectors that arise when splitting up the vector of the page according to the syntax

$$\textit{vector} ::= \textit{timestamp contents}$$

The *reference* of a Logiweb page is the first of the two bit vectors that arise when splitting up the page according to the syntax

$$\mathcal{P} ::= \textit{reference contents}$$

Hence, the reference equals the concatenation of the version, the key, and the timestamp of the page.

The *bibliography* and *body* of a Logiweb page are the bit vectors that arise when splitting up the contents of the page according to the syntax

$$\textit{contents} ::= \textit{bibliography body}$$

The relation between the various concepts may be displayd thus:

$$\mathcal{P} ::= \overbrace{\textit{version key timestamp}}^{\textit{reference}}\ \overbrace{\textit{bibliography body}}^{\textit{contents}}$$
$$\mathcal{P} ::= \textit{version key}\ \underbrace{\textit{timestamp bibliography body}}_{\textit{vector}}$$

A Logiweb page is *locally valid* if the following holds:

- The key of the page equals the Logiweb-160 global hash key of the vector of the page.
- All timestamps in the bibliography are positive.
- The timestamp of the page is greater than each timestamp in the bibliography.

As can be seen, the version of any valid Logiweb page is 0000 0001. When and if RIPEMD-160 is cracked by hackers or it otherwise becomes easy to generate RIPEMD-160 collisions, the definition of Logiweb pages has to be extended in a backward compatible manner so that any page that conforms to some, future standard must have versions different from 0000 0000 and 0000 0001. As long as RIPEMD-160 serves its purpose of generating unique keys, it is hard to imagine a purpose for changing the definition of Logiweb pages.

3.9 Logiweb webs

A *Logiweb web* is a finite, non-empty sequence of Logiweb pages. A Logiweb web is *consistent* if it satisfies the following properties:

- Every page in the web is locally valid.
- Any two pages in the web that have equal references also have equal vectors.
- For every reference in every bibliography in the web, there exists a page in the web with that reference.

The requirement that every page must be locally valid implies that the timestamp of each page must be greater than all timestamps in the bibliography of that page. This rules out circular bibliographic references. Hence, the pages of a web and their bibliographic references form a directed, acyclic graph. Since, furthermore, every Logiweb web is required to contain at most finitely many pages. This ensures that every web is *well-founded* in the sense that there are no infinite sequences $(p_0, p_1, \ldots)$ such that p_i refers to p_{i+1} for all natural numbers i. For this reason it is possible to prove properties about pages of a web by induction in their bibliographic references.

The requirement that any two pages with equal references have equal vectors ensures that every web defines a function from references to vectors. In other words, given a web, and given the version, key, and timestamp of a page in the web, it is possible to look up the bibliography and body of the page. The bibliography and body will be uniquely determined even if the web contains several copies of the given page.

3.10 Logiweb addresses

Whenever a Logiweb node sends a message, it sends it to some Logiweb address. Since Logiweb can be supported by several host protocols, a Logiweb address must identify the host protocol and, furthermore, identify the exact address in some host protocol specific format.

At present, Logiweb is supported by the host protocols UDP and TCP, which in turn are hosted by IP version 4. At present, any Logiweb address is eight bytes long. If a Logiweb node sends a message to a given Logiweb address, then the first two bytes of the address indicates which host protocol (UDP/IPv4 or TCP/IPv4) to use, the next two bytes indicates which port to send to, and the next four bytes indicates the IP address to send to.

The syntax of a Logiweb address is:

address	::=	*udp_ip_v4_address* | *tcp_ip_v4_address*
udp_ip_v4_address	::=	*family udp_ip_v4_id port ip*
family	::=	0000 0001
udp_ip_v4_id	::=	0000 0000
tcp_ip_v4_address	::=	*family tcp_ip_v4_id port ip*
tcp_ip_v4_id	::=	0000 0001
port	::=	*portlsb portmsb*
portlsb	::=	*byte*
portmsb	::=	*byte*
ip	::=	$byte^4$

As an example of an address, udp-port 5 of a network interface with ip address 1.2.3.4 will have the following Logiweb address:

```
0000 0001   0000 0000   0000 0101   0000 0000
0000 0001   0000 0010   0000 0011   0000 0010
```

Each Logiweb node must have at least one Logiweb address. A Logiweb node may have more than one Logiweb address. As an example, a Logiweb node may have both a UDP and a TCP address. As another example, a Logiweb node may have more than one connection to the internet.

4 Representation by Logiweb binary trees

4.1 Introduction

Within Logiweb, many quantities are represented by Logiweb binary trees. The following paragraphs present some of the representations.

4.2 Truth values

Truth and falsehood are represented by T and (T, T), respectively. Let F denote (T, T). Now truth and falsehood are represented by T and F, respectively.

T and F will be referred to as the *canonical representation* of truth and falsehood. In addition to the canonical representation we also introduce a *liberal representation.* In the liberal representation, T represents truth whereas all Logiweb binary trees of form (x, y) represent falsehood.

4.3 Bits

The *binary terms* (*bits*) 0 and 1 are canonically represented by T and (T, T), respectively. T liberally represents 0 whereas all binary trees of form (x, y) represent 1.

Note that 0 and T are represented by the same binary tree. This is in line with recursion theory in general and Gödels 1931 paper in particular. Note that the convention is opposite of e.g. the C programming language.

4.4 Pairs

We shall use $x :: y$ as an alternative notation for (x, y). $x :: y$ is right associative such that e.g. $x :: y :: z$ means $x :: (y :: z)$.

4.5 Tuples

A tuple $\langle a_1, \ldots, a_n \rangle$ is represented by $a_1 :: \cdots a_n :: \mathsf{T}$. As an example, the empty tuple $\langle\,\rangle$ is represented by T.

4.6 Bitvectors

Bit vectors are represented as tuples of bits. In tuples of bits, bit number zero is the first bit in the tuple. When written as sequences of bits, bit zero is written rightmost. Hence, when writing out a tuple of bits, one has to reverse the order of the bits. As an example, the bit vector

$$(01100)_{\mathrm{bit}}$$

is represented by the tuple

$$\langle 0, 0, 1, 1, 0 \rangle$$

As examples, the empty bit vector $(\,)_{\mathrm{bit}}$ is represented by T and the one element bit vector $(\mathsf{T})_{\mathrm{bit}}$ is represented by $\langle \mathsf{T} \rangle = (\mathsf{T}, \mathsf{T})$.

4.7 Bytes

A byte (eight bit word) is represented as an 8-bit bit vector.

4.8 Characters

Characters in ASCII / ISO Latin 1 are represented as bytes. As an example,

$$\mathrm{A}_{\mathrm{char}}$$

is represented by

$$(00100001)_{\mathrm{bit}}$$

which in turn is represented by

$$\langle 1, 0, 0, 0, 0, 1, 0, 0 \rangle$$

In the binary number representation, the bitvector above represents the number 65.

4.9 Strings

A string is represented as a tuple of characters. The first character in the tuple represents the first character in the string. As an example

“AB”

is represented by

$$\langle \mathrm{A}_{\mathrm{char}}, \mathrm{B}_{\mathrm{char}} \rangle$$

4.10 Binary number representation

In the binary number representation, a bit vector represents a natural number. When written as a tuple, the least significant bit is the first bit in the tuple. When written as a bit vector, the least significant bit is rightmost. As examples,

$$(01100)_{\mathrm{bit}}$$

and

$$\langle 0, 0, 1, 1, 0 \rangle$$

both represent twelve in the binary representation. For all natural numbers n we shall use $\mathrm{binary}(n)$ to denote the shortest bitvector that represents the number. As examples, $\mathrm{binary}(12) = \langle 0, 0, 1, 1 \rangle$ and $\mathrm{binary}(0) = \mathsf{T}$.

4.11 Small natural numbers

When size is a primary concern, small natural numbers will be represented using a rather complicated scheme that is represented in the following:

The Polish prefix representation of a Logiweb binary tree x is a sequence of bits. Let $\mathrm{ord}(x)$ denote the value of that sequence when it is interpretted as a binary number with the least significant bit rightmost (i.e. opposite of the binary number representation). The following table gives some examples of values of $\mathrm{ord}(x)$:

x	prefix representation	$\mathrm{ord}(x)$
T	0	0
(T,T)	100	4
$(\mathsf{T},(\mathsf{T},\mathsf{T}))$	10100	20
$((\mathsf{T},\mathsf{T}),\mathsf{T})$	11000	24
$(\mathsf{T},(\mathsf{T},(\mathsf{T},\mathsf{T})))$	1010100	84

If the binary trees x and y are different, then $\mathrm{ord}(x)$ and $\mathrm{ord}(y)$ are also different, so $\mathrm{ord}(x)$ determines x uniquely. We shall refer to $\mathrm{ord}(x)$ as the *ordinate* of x.

Let $\#0$ denote the binary tree that has the smallest ordinate, i.e. $\#0 \doteq \mathsf{T}$. Let $\#1$ denote the binary tree that has the second smallest ordinate, i.e. $\#1 \doteq (\mathsf{T},\mathsf{T})$. Let $\#2$ denote the binary tree that has the third smallest ordinate, i.e. $\#2 \doteq (\mathsf{T},(\mathsf{T},\mathsf{T}))$ and so on.

With these definitions, $\#0, \#1, \#2, \ldots$ is an enumeration of all binary trees.

When size is important and i is a small, natural number, we shall use $\#i$ to represent the number i.

4.12 Digits

We shall use small numbers to represent the digits 0 to 9.

The following table displays the definitions of the ten digits.

$$\begin{array}{lcl}
0_{\mathrm{digit}} & \doteq & \mathsf{T} \\
1_{\mathrm{digit}} & \doteq & (\mathsf{T},\mathsf{T}) \\
2_{\mathrm{digit}} & \doteq & (\mathsf{T},(\mathsf{T},\mathsf{T})) \\
3_{\mathrm{digit}} & \doteq & ((\mathsf{T},\mathsf{T}),\mathsf{T}) \\
4_{\mathrm{digit}} & \doteq & (\mathsf{T},(\mathsf{T},(\mathsf{T},\mathsf{T}))) \\
5_{\mathrm{digit}} & \doteq & (\mathsf{T},((\mathsf{T},\mathsf{T}),\mathsf{T})) \\
6_{\mathrm{digit}} & \doteq & ((\mathsf{T},\mathsf{T}),(\mathsf{T},\mathsf{T})) \\
7_{\mathrm{digit}} & \doteq & ((\mathsf{T},(\mathsf{T},\mathsf{T})),\mathsf{T}) \\
8_{\mathrm{digit}} & \doteq & (((\mathsf{T},\mathsf{T}),\mathsf{T}),\mathsf{T}) \\
9_{\mathrm{digit}} & \doteq & (\mathsf{T},(\mathsf{T},(\mathsf{T},(\mathsf{T},\mathsf{T}))))
\end{array}$$

4.13 Decimal number representation

In the decimal number representation we represent a natural number as a list of digits with the least significant digit first. As an example

$$\langle 3_{\mathrm{digit}}, 2_{\mathrm{digit}}, 1_{\mathrm{digit}} \rangle$$

represents the number 123. We shall use decimal(n) to denote the shortest list of digits that represents the number n. As an example decimal(123) = $\langle 3_{\mathrm{digit}}, 2_{\mathrm{digit}}, 1_{\mathrm{digit}} \rangle$.

4.14 Sign magnitude representation

In the sign magnitude representation, we represent an integer as a pair whose elements represent the sign and magnitude, respectively, of the integer. A plus sign is represented as T and a minus sign as F. As an example,

$$\langle \mathsf{F}, 3_{\mathrm{digit}}, 2_{\mathrm{digit}}, 1_{\mathrm{digit}} \rangle$$

represents -123. The number zero can be represented by an arbitrary sign followed by an arbitrary number of zero digits. We make the additional convention that T represents zero in the sign magnitude representation, so that zero can be represented as the empty tuple.

We shall use signmag(n) to denote the shortest tuple that represents the integer n in the sign magnitude representation. Some examples read:

$$\begin{array}{lcl} \mathrm{signmag}(123) & = & \langle \mathsf{T}, 3_{\mathrm{digit}}, 2_{\mathrm{digit}}, 1_{\mathrm{digit}} \rangle \\ \mathrm{signmag}(0) & = & \mathsf{T} \\ \mathrm{signmag}(-123) & = & \langle \mathsf{F}, 3_{\mathrm{digit}}, 2_{\mathrm{digit}}, 1_{\mathrm{digit}} \rangle \end{array}$$

4.15 Decimal fractions

We shall represent decimal fractions as three element tuples $\langle m, e, p \rangle$ where m and e are sign magnitude representations of integers M and E, and p represents a truth value. In the decimal fraction representation, $\langle m, e, p \rangle$ represents the value

$$M \cdot 10^{E}$$

regardless of the value of p.

We shall refer to m, e, and p as the mantissa, exponent, and exactness, respectively, of the decimal fraction. The exactness has no influence on the value that the tuple represents, but it does have effect on operations such as the plus operation. To be more specific, the exactness affects whether or not the plus operation involves rounding of the result.

A decimal fraction is said to be an *exact fraction* if its exactness is T and is said to be a *floating fraction* if its exactness represents falsehood.

We shall refer to the number of digits of the mantissa of a floating fraction as the *precision* of the floating fraction. When counting the digits, all digits count, even zero digits at the beginning and end of the mantissa. As an exception, if the mantissa has no digits (in which case the mantissa represents zero) then the precission is 1 by convention.

Two decimal fractions are added thus: If both are exact, they are just added with no rounding. As an example, $10^{100} + 10^{-100}$ has 201 digits. If both are floating and if they have different precision, the result is an exception. If the decimal fractions are floating and have the same precision, then they are added as exact fractions and after that the result is rounded to the common precision of the two floating fractions. If one of the decimal fractions is exact and the other is floating, then they are added as exact fractions and after that the result is rounded to the precision of the floating fraction.

We shall use the prefix `F` to denote that a decimal fraction is floating. As an example

$$1.23\texttt{F}$$

may be represented thus:

$$\langle\langle \mathsf{T}, 3_{\mathrm{digit}}, 2_{\mathrm{digit}}, 1_{\mathrm{digit}}\rangle, \langle \mathsf{F}, 2_{\mathrm{digit}}\rangle, \mathsf{F}\rangle$$

4.16 Symbols

As mentioned in Section 3.8, Logiweb references are bit vectors. A Logiweb reference is represented like any other bit vector as described in Section 4.6.

A symbol is a pair $p :: i$ where p and i are referred to as the *page* and *id* of the symbol, respectively. The page of a symbol must be T or a Logiweb reference. The id can be any Logiweb binary tree.

Logiweb symbols correspond to Lisp atoms and to Common Lisp symbols. A Logiweb symbol with page p and id i corresponds to a Common Lisp symbol with name i that occurs in package p. Contrary to Common Lisp, a Logiweb symbol has one and only one page whereas a Common Lisp symbol may occur in zero, one or more packages. Also contrary to Common Lisp, the name i of a Logiweb symbol is a rather unreadable Logiweb binary tree whereas the name of a Common Lisp symbol is a more readable string of characters.

Symbols whose page is T are referred to as *predefined symbols*. Predefined symbols have fixed meanings as specified later.

Symbols whose page is a Logiweb reference are referred to as *defined symbols*. Given a Logiweb web W, the meaning of a defined symbol $p :: i$ relative to W is defined by the page referred to by p. The details are specified later.

NOTE: It seems wiser to let symbols have form $\langle p, i_1, \ldots, i_q\rangle$. That allows indefinite extension so that e.g. a page Decimal could have two infinities of symbols:

$$\langle \mathrm{Decimal}, \mathrm{Exact}, \mathrm{mantisa}, \mathrm{exponent}\rangle$$

and

$$\langle \mathrm{Decimal}, \mathrm{Float}, \mathrm{mantisa}, \mathrm{exponent} \rangle$$

One can then look up the meaning of a symbol by successively taking in more and more of the symbol until a meaning is found (e.g. $\langle \mathrm{Decimal} \rangle$ has no meaning, $\langle \mathrm{Decimal}, \mathrm{Float} \rangle$ has. However, this way of looking at symbols does not contradict the definition above, so why bother too much.

4.17 Predefined symbols

Define

$$b_i \doteq (\mathsf{T}, \#i)$$

where $\#i$ was defined in Section 4.11. The sequence $b_0, b_1, b_2, \ldots$ constitutes an enumeration of all predefined symbols. Now define

T	=	b_0	Computable constructs
P	=	b_1	
A	=	b_2	
S	=	b_3	
K	=	b_4	
Stderr	=	b_5	Output event identifiers
Exec	=	b_6	
Extend	=	b_7	
Boot	=	b_8	Input event identifiers
Interrupt	=	b_9	
Stdin	=	b_{10}	
Extended	=	b_{11}	
Exit	=	b_{12}	Interrupt identifiers
Watchdog	=	b_{13}	
Memory	=	b_{14}	
X	=	b_{15}	Codex inconsistency
Codex	=	b_{16}	Second level indices
Subcache	=	b_{17}	
Parsetree	=	b_{18}	
Page	=	b_{19}	
Pagename	=	b_{20}	Aspects
Unpack	=	b_{21}	
Symbolname	=	b_{22}	
Parameters	=	b_{23}	
TeXname	=	b_{24}	
Codify	=	b_{25}	
Value	=	b_{26}	

4.18 Terms

A Logiweb *term* is a list with at least one element whose first element is a symbol and whose remaining elements (if any) are terms.

Logiweb terms correspond to Lisp S-expressions.

In a Logiweb term $\langle r, t_1, \ldots, t_n \rangle$, r will be referred to as the *root* of the term and t_i will be referred to as the i'th *subterm* of t.

4.19 Association lists

An *association list* is a list

$$\langle k_1 :: v_1, \ldots, k_n :: v_n \rangle$$

where n is a natural number, where $k_1, \ldots, k_n$ and $v_1, \ldots, v_n$ are Logiweb binary trees, where $v_1, \ldots, v_n$ all differ from T, and where

$$\operatorname{ord}(k_1) < \operatorname{ord}(k_2) < \cdots < \operatorname{ord}(k_n)$$

In the association list, $k_1, \ldots, k_n$ are referred to as *keys* and $v_1, \ldots, v_n$ are referred to as *values*.

As no two keys in an association list are equal, an association list a defines a function from keys to values.

The ordering imposed on keys is completely arbitrary. Any total ordering would do. The purpose of the ordering is to allow off stage implementation of association lists in a way in which the ordering of associations is lost. As an example, one may implement association lists as binary search trees behind the scenes. As long as one merely access the association list using the functions described in the following, access will happen efficiently. The moment one takes the head of the association list, the system will have to convert the off stage representation back to a genuine association list which is handed to the "head" function. When doing this conversion, the convention above allows to sort the associations in a well defined order.

4.20 Association list access functions

For all association lists a and for all Logiweb binary trees k, let

$$\langle a|k \rangle$$

denote the unique Logiweb binary tree v for which $k :: v$ belongs to the list a if such a v exists, and let $\langle k|a \rangle$ denote T otherwise.

We shall use $\langle a|k_1, \ldots, k_q \rangle$ to denote $\langle a|\langle k_1, \ldots, k_q \rangle\rangle$.

For all association lists a and for all Logiweb binary trees k and v, let

$$\langle a|k{:=}v \rangle$$

denote the unique association list a' for which $\langle a'|k \rangle = v$ and which otherwise coincides with a. Note that $\langle a|k{:=}v \rangle$ may add or change an association when $v \neq \mathsf{T}$ and may remove an association when $v = \mathsf{T}$.

For all lists p and q let $p \cdot q$ denote p concatenated with q.

For all association lists a and for all Logiweb binary trees p let

$$\text{trie\_get}(p, a)$$

denote the unique association list a' for which

$$\langle a' | q \rangle = \langle a | p \cdot q \rangle$$

For all association lists a and v and for all Logiweb binary trees k let

$$\text{trie\_put}(k, v, a)$$

denote the unique association list a' for which $\text{trie\_get}(k, a') = v$ and which otherwise coincides with a.

4.21 Association list merge function

For all Logiweb binary trees x and y define $\text{value\_merge}(x, y)$ thus:

$$\begin{array}{lcll}
\text{value\_merge}(x, y) & = & x & \text{if } y = \mathsf{T} \\
\text{value\_merge}(x, y) & = & y & \text{if } x = \mathsf{T} \\
\text{value\_merge}(x, y) & = & x & \text{if } x = y \\
\text{value\_merge}(x, y) & = & \mathbf{X} & \text{otherwise}
\end{array}$$

For all association lists a and a' define $a \oplus a'$ such that

$$\langle a \oplus a' | v \rangle = \text{value\_merge}(\langle a | v \rangle, \langle a' | v \rangle)$$

4.22 Codices

A Logiweb *codex* is an association list c for which all keys have form $\langle p, s, a \rangle$ where p is a Logiweb page reference and s and a are symbols.

We shall refer to $\langle c | p, s, a \rangle$ as the *aspect* a of the symbol s as defined on page p w.r.t. the codex c.

As an example, $\langle c | s \text{ head}, s, \text{Value} \rangle$ denotes the Value aspect of the symbol s as defined on the home page of s w.r.t. the codex c. This corresponds to (symbol-function s) in Common Lisp when s denotes a Common Lisp function.

As another example, $\langle c | s \text{ head}, s, \text{Macro} \rangle$ denotes the Macro aspect of the symbol s as defined on the home page of s w.r.t. the codex c. This corresponds to (symbol-function s) in Common Lisp when s denotes a Common lisp macro.

As a third example, $\langle c | s \text{ head}, s, \text{Name} \rangle$ denotes the Name aspect of the symbol s as defined on the home page of s w.r.t. the codex c. This corresponds to (symbol-name s) in Common Lisp.

More generally, $\langle c|s \text{ head}, s, a\rangle$ corresponds to the property a of the property list of the symbol s in Common Lisp. Furthermore, a Logiweb codex corresponds to a Common Lisp *environment*.

If $p \neq (s \text{ head})$ then $\langle c|p, s, a\rangle$ has no immediate correspondence in Common Lisp.

4.23 Lambda terms

We shall represent computable functions using the computable part of Map Theory. Expressed in the framework of lambda calculus, the syntax $\bar{\mathcal{T}}$ of the computable part of Map Theory reads:

$$\begin{array}{lcl} \mathcal{V} & ::= & x \mid y \mid z \mid \cdots \\ \bar{\mathcal{T}} & ::= & \mathcal{V} \mid \lambda\mathcal{V}.\bar{\mathcal{T}} \mid \bar{\mathcal{T}}\bar{\mathcal{T}} \mid \mathsf{T} \mid \text{if}(\bar{\mathcal{T}},\bar{\mathcal{T}},\bar{\mathcal{T}}) \end{array} \qquad \text{Lambda terms}$$

The reduction rules of the computable part of Map Theory read:

$$\begin{array}{lcl} (\lambda x.y)z & \to & \langle y \mid x := z\rangle \\ \mathsf{T}z & \to & \mathsf{T} \\ \text{if}(\mathsf{T}, y, z) & \to & y \\ \text{if}(\lambda u.v, y, z) & \to & z \end{array} \qquad \text{Lambda reductions}$$

Above, $\langle y \mid x := z\rangle$ denotes the term y in which the variable x is replaced by the term z with suitable renaming of bound variables to avoid variable clashes.

The computable part of Map Theory uses normal order reduction and reduction proceeds until a term is on *root normal form.* A term is on root normal form when it has one of the forms T or $\lambda x.y$.

A term is said to be on *atom normal form* if it is identical to the term T. A term is said to be on *function normal form* if it is on root normal form without being on atom normal form. Hence, a term is on function normal form when it is of form $\lambda x.y$. With these conventions, the reduction rules may be stated thus:

$$\begin{array}{lcll} (\lambda x.y)z & \to & \langle y \mid x := z\rangle & \\ \mathsf{T}z & \to & \mathsf{T} & \\ \text{if}(x, y, z) & \to & y & \text{if } x \text{ is on atom normal form} \\ \text{if}(x, y, z) & \to & z & \text{if } x \text{ is on function normal form} \end{array}$$

4.24 Combinator terms

On a computer, however, it is inconvenient to reduce terms that are expressed in a syntax that includes variables. This is so because the operation $\langle y \mid x := z\rangle$ is complicated. For that reason, the engine of the Logiweb machine implements a combinatorial version of the computable part of Map Theory. In the combinatorial version, terms are expressed using T plus the following three combinators:

$$\begin{array}{lcl} \mathsf{S} & \doteq & \lambda x.\lambda y.\lambda z.xz(yz) \\ \mathsf{K} & \doteq & \lambda x.\lambda y.x \\ \mathsf{P} & \doteq & \lambda x.\lambda y.\lambda z.\text{if}(z, x, y) \end{array} \qquad \text{Combinators}$$

Using these combinators, the syntax $\mathcal{T}$ of terms may be written thus:

$$\mathcal{T} \quad ::= \quad \mathcal{T}\mathcal{T} \mid \mathsf{S} \mid \mathsf{K} \mid \mathsf{T} \mid \mathsf{P} \qquad \text{Combinator terms}$$

The reduction rules may be stated thus:

$$\begin{array}{ll}
\mathsf{S}xyz & \to xz(yz) \\
\mathsf{K}xy & \to x \\
\mathsf{T}x & \to \mathsf{T} \\
\mathsf{P}xy\mathsf{T} & \to x \\
\mathsf{P}xy(\mathsf{S}) & \to y \\
\mathsf{P}xy(\mathsf{S}u) & \to y \\
\mathsf{P}xy(\mathsf{S}uv) & \to y \\
\mathsf{P}xy(\mathsf{K}) & \to y \\
\mathsf{P}xy(\mathsf{K}u) & \to y \\
\mathsf{P}xy(\mathsf{P}) & \to y \\
\mathsf{P}xy(\mathsf{P}u) & \to y \\
\mathsf{P}xy(\mathsf{P}uv) & \to y
\end{array} \qquad \text{Combinator reductions}$$

The engine still reduces until root normal form. In the combinator formulation, a term is on root normal form when it has one of the following forms:

$$\begin{array}{llll}
\mathsf{S} & \mathsf{K} & \mathsf{T} & \mathsf{P} \\
\mathsf{S}u & \mathsf{K}u & & \mathsf{P}u \\
\mathsf{S}uv & & & \mathsf{P}uv
\end{array} \qquad \text{Root normal forms}$$

A term is still said to be on atom normal form if it is identical to the term T and a term is still said to be on function normal form if it is on root normal form without being on atom normal form. Hence, a term is on function normal form if it has one of the following forms:

$$\begin{array}{lll}
\mathsf{S} & \mathsf{K} & \mathsf{P} \\
\mathsf{S}u & \mathsf{K}u & \mathsf{P}u \\
\mathsf{S}uv & & \mathsf{P}uv
\end{array} \qquad \text{Function normal forms}$$

With these conventions, the reduction rules may be stated thus:

$$\begin{array}{lll}
\mathsf{S}xyz & \to xz(yz) & \\
\mathsf{K}xy & \to x & \\
\mathsf{T}x & \to \mathsf{T} & \\
\mathsf{P}xyz & \to x & \text{if } z \text{ is on atom normal form} \\
\mathsf{P}xyz & \to y & \text{if } z \text{ is on function normal form}
\end{array}$$

The reduction order in the combinator version is thus:

$$\begin{array}{llll}
\mathsf{S}xyzu_1\cdots u_n & \to & xz(yz)u_1\cdots u_n & \\
\mathsf{K}xyu_1\cdots u_n & \to & xu_1\cdots u_n & \\
\mathsf{T}xu_1\cdots u_n & \to & \mathsf{T}u_1\cdots u_n & \\
\mathsf{P}xyzu_1\cdots u_n & \to & xu_1\cdots u_n & \text{if } z \text{ is on atom normal form} \\
\mathsf{P}xyzu_1\cdots u_n & \to & yu_1\cdots u_n & \text{if } z \text{ is on function normal form}
\end{array}$$

If z is neither on atom nor on function normal form then reduction of

$$\mathsf{P}xyzu_1\cdots u_n$$

leads to reduction of z. This last requirement has the consequence that the reduction order of the combinator version differs from normal order reduction. We shall say that the reduction order is *call by need.*

4.25 The value function

For all terms t, symbols s, and codices c we shall define the value vct of the term t and the value Vcs of the symbol s w.r.t. the codex c as follows:

$$\begin{array}{lcl}
vc\langle s, t_1, \ldots, t_n\rangle & = & Vcs(vct_1)\ldots(vct_n) \\
Vcs & = & vc\langle c|s \text{ head}, \mathbf{Codex}, s, \text{Value}\rangle \\
 & & \quad \text{if } s \notin \{\mathbf{A}, \mathbf{S}, \mathbf{K}, \mathbf{T}, \mathbf{P}\} \\
Vc\mathbf{A} & = & \mathsf{SKK} \\
Vc\mathbf{S} & = & \mathsf{S} \\
Vc\mathbf{K} & = & \mathsf{K} \\
Vc\mathbf{T} & = & \mathsf{T} \\
Vc\mathbf{P} & = & \mathsf{P}
\end{array}$$

For completeness we define $vc\mathsf{T} = \mathsf{T}$ so that vct and Vcs are defined for all Logiweb binary trees c, t, and s.

4.26 The boot representation

The *boot representation* allows to represent computable functions by logiweb binary trees in a particularly simple way.

The boot representation is defined by a function b. Given a Logiweb binary tree t, bt equals the computable function represented by t. The function b and the auxiliary function (t last) are defined thus:

$$\begin{array}{lcll}
bt & = & \mathsf{T} & \text{if } t = \mathsf{T} \text{ else} \\
bt & = & v\mathsf{T}t & \text{if } v \text{ head} = \mathbf{A} \text{ else} \\
bt & = & b(t \text{ tail last}) & \\
t \text{ last} & = & t \text{ head} & \text{if } t \text{ tail} = \mathsf{T} \\
t \text{ last} & = & t \text{ tail last} & \text{otherwise}
\end{array}$$

To represent a computable function in the boot representation, one can represent the computable function by a term of syntax $\mathcal{T}$, and then one can represent the term as a Logiweb binary tree thus:

xy	is represented by	$\langle \mathbf{A}, x, y\rangle$
S	is represented by	$\langle \mathbf{S}\rangle$
K	is represented by	$\langle \mathbf{K}\rangle$
T	is represented by	$\langle \mathbf{T}\rangle$
P	is represented by	$\langle \mathbf{P}\rangle$

The definition of bt says that b recursively descends to the rightmost subterm of t until it finds an $\mathbf{A}$. This allows to construct a Logiweb binary tree that represents a mathematical text and which, in its last appendix, contains a computable function expressed in the boot representation.

4.27 The eval function

The eval function e takes three arguments: a codex c, a *message* m , and a term t. A message m consists of an *aspect* a followed by *submessages* $m_1, \ldots, m_p$.

We shall refer to $ecmt$ as the result of sending the message m to the term (or object) t.

The definition of the eval function e reads:

$$\begin{array}{lcll} ecm & = & vc & \text{if } m \text{ head} = \text{Value} \\ ecm & = & b & \text{if } m \text{ head} = \text{Boot} \\ ecm & = & vcmc & \text{otherwise} \end{array}$$

5 Unpacking of pages

5.1 Logiweb caches

For all Logiweb webs W and all page references r let page(r, W) denote the page in W referenced by r if such a page exists and let page$(r, W) = \mathsf{T}$ otherwise. If p is a page then let Ref(p), Data(p), and Bib(p) denote the reference, data, and bibliography, respectively, of p.

Given a Logiweb reference r, the task of a Logiweb browser is to fetch and "unpack" the Logiweb page page(r, W) where W is the web maintained by all connected Logiweb servers at that given moment. Unpacking a page allows the browser to extract definitions, executable code, theorems, fonts, and many other data structures from the page, and ultimately allows the browser to display the information to a user.

Among other, unpacking a page produces a *parse tree* and a *codex*.

A codex is a set of definitions. A codex can define values, functions, executable code, theorems, proofs, fonts, and anything else that a computer can manipulate.

The parse tree of a page is a representation of the page which is particularly suited for displaying and editing the page, and it is also the basis for finding the codex of the page. To display a page, a browser has to do "reverse parsing" of the parse tree. This is in contrast to ordinary programming languages in which the human readable form of a program is "parsed" to obtain the parse tree. In Logiweb, the parse tree is the primary form which has to be reverse parsed to give a human readable form.

To unpack $p = $ page(r, W), the browser first has to unpack all pages $q_1, \ldots, q_n$ in the bibliography of p; and before that the browser must unpack all pages in the bibliographies of $q_1, \ldots, q_n$ and so on. Hence, to unpack a page p, the browser first has to unpack all pages reachable through one or more bibliographic references starting at p.

A Logiweb browser typically needs to access the same pages over and over again, and for that reason it is convenient for a browser to maintain a *Logiweb cache* of pages. A Logiweb cache is an association list whose keys have form $r :: r'$ where r must be a page reference. Unpacking a page page(r, W) results in a number of different pieces of information, all of which are stored under keys of form $r :: r'$.

Now suppose W is a Logiweb web and that b is a list of page references. To unpack all pages referenced by b, a Logiweb browser has to construct a Logiweb cache $C(b, W)$ which contains information about all pages referenced by b

together with all pages reachable from those pages through one or more bibliographic references. The definition of $C(b, W)$ defines the notion of unpacking. The definition of $C(b, W)$ reads:

$$\begin{array}{lcl}
C(\mathsf{T}, W) & = & \mathsf{T} \\
C(r :: b, W) & = & C_1(r, W) \oplus C(b, W) \\
C_1(r, W) & = & \mathrm{let}\, p = \mathrm{page}(r, W)\, \mathrm{in} \\
& & \mathrm{let}\, s = C(\mathrm{Bib}(p), W)\, \mathrm{in} \\
& & \mathrm{let}\, t = \mathrm{parse}(p, s)\, \mathrm{in} \\
& & \mathrm{let}\, x = \mathrm{codex}(t, s)\, \mathrm{in} \\
& & \langle s| \\
& & \quad r, \mathbf{Page} := p| \\
& & \quad r, \mathbf{Subcache} := s| \\
& & \quad r, \mathbf{Parsetree} := t| \\
& & \quad r, \mathbf{Codex} := x\rangle
\end{array}$$

The functions codex(t, c) and parse(p, c) are described in the following.

We shall refer to $C_1(r, W)$ as the *minimal cache* of the reference r w.r.t. the web W. Given a reference r and a cache c that contains r, the minimal cache of r equals $C_2(r, c)$ which is defined thus:

$$\begin{array}{lcl}
\boxed{C_2(r, c)} & = & \mathrm{let}\, p = \langle c|r, \mathbf{Page}\rangle\, \mathrm{in} \\
& & \mathrm{let}\, s = \langle c|r, \mathbf{Subcache}\rangle\, \mathrm{in} \\
& & \mathrm{let}\, t = \langle c|r, \mathbf{Parsetree}\rangle\, \mathrm{in} \\
& & \mathrm{let}\, x = \langle c|r, \mathbf{Codex}\rangle\, \mathrm{in} \\
& & \langle s| \\
& & \quad r, \mathbf{Page} := p| \\
& & \quad r, \mathbf{Subcache} := s| \\
& & \quad r, \mathbf{Parsetree} := t| \\
& & \quad r, \mathbf{Codex} := x\rangle
\end{array}$$

5.2 The parse tree transformation

Suppose a Logiweb page p has reference r, bit-vector d and bibliography $b = \langle b_1, \ldots, b_n\rangle$. The parse tree transformation parse(p, c) inspects

$$u = \langle c|b_n, \mathbf{Codex}, b_n :: \mathsf{T}, \mathrm{Unpack}\rangle$$

If u differs from T then parse(p, c) returns *vcup*.

If u equals T then the transformation does a "default parse tree transformation" as follows: First, the transformation converts d from Polish prefix representation to a Logiweb binary tree x. Then the transformation does page references substitution as follows:

Page reference substitution in a Logiweb binary tree x proceeds thus: Write x on form $\langle q :: i, x_1, \ldots, x_n\rangle$. Then do page reference substitution in $x_1, \ldots, x_n$. Then interpret q as a small, natural number. If $q = 0$ then replace q by r. If $0 < q \le n$ then replace q by the b_q. If $n < q$ then replace q by T.

5.3 The codex transformation

The codex transformation has the following property:

$$\text{codex}(\langle \text{Definition}, \langle s, \ldots \rangle, \langle a, \ldots \rangle, v \rangle, c) = \langle \mathsf{T} | \langle s, a \rangle {:=} v \rangle$$

Now suppose $t = \langle s, t_1, \ldots, t_n \rangle$ where $s \neq \text{Definition}$. The codex transformation $\text{codex}(t, c)$ inspects

$$u = \langle c | s\ \text{head}, \mathbf{Codex}, s, \text{Codify} \rangle$$

If $u \neq \mathsf{T}$ then $\text{codex}(t, c)$ returns *ecut*. If $u = \mathsf{T}$ then $\text{codex}(t, c)$ does a "default codex transformation", i.e. it returns $\text{codex}(t_1, c) \oplus \cdots \oplus \text{codex}(t_n, c)$.

5.4 Default unpacking

By default, a page is unpacked by a default parse tree transformation followed by a default codex transformation. One may override the default parse tree transformation by defining an unpacker $\langle c | r, \mathbf{Codex}, r :: \mathsf{T}, \text{Unpack} \rangle$ in the last bibliographic reference r and one may override the default codex transformation by placing a symbol s at the root of the parse tree for which $\langle c | s\ \text{head}, \mathbf{Codex}, s, \text{Codify} \rangle \neq \mathsf{T}$.

The intension is that default unpacking should be used for unpacking a few, fundamental Logiweb pages such as pages that define non-standard unpacking. Hence, default unpacking is intended for bootstrapping the system, not for general use.

As examples, custom parse tree transformations may be used for decompressing compressed pages and decrypting encrypted pages.

As another example, if one defines e.g. a Logiweb font metric (lfm) file format and one wants to convert e.g. a TEX font metric (tfm) file to that format, then it is possible to use the tfm file verbatim as the data part of a Logiweb page as long as the unpacker for that page converts tfm to lfm format. If, furthermore, the Logiweb server represents Logiweb pages as a reference, a bibliography, and a link to a file that contains the data, then it is possible to let the Logiweb server and a TEX system on the same host share the tfm file.

5.5 Presentation of pages

Given a reference r and a web W, one may *present* the page of W referenced by r. A "presentation" is something a human being can absorb. A "presentation" is typically an image on a screen or a print on paper, but it can also be other things such as an electronic voice reading the page.

Contrary to "$\underline{\text{re}}$presentations", a presentation may not capture all information about a page. In general, it will be impossible to reconstruct a page from a presentation. Actually, good presentations typically suppress information that is considered irrelevant.

Presentations may be interactive. As an example, a screen presentation may respond to mouse clicks and keystrokes.

A page will typically have many presentations. As an example, a page may have one presentation suited for display on a screen, one suited for printing, one suited for editing, and so on.

Presentations are identified by messages. Hence, to present a page, one needs a web W, a page reference r, and a message m which indicates how the page should be presented.

Given a web W, a page reference r, and a message m, let c be the minimal cache that contains r and also contains all pages referenced from m. The presentation of r specified by m is now given by $f = ecmr$. The function f is then used as initial function in a Logiweb machine.

How output events from the Logiweb machine are interpretted depends on the browser in use. It is also the browser that supplies input events to the Logiweb machine. The browser is typically defined by the boot page (see later).

Two particularly simple presentations are predefined, however. They are called the Lisp form and the TEX form, respectively. Both forms present a page as a string of Ascii characters ("newline" plus characters in the range from 32 to 127). The Lisp form is an S-expression suited for manipulation by Lisp programs. The TEX form is suited for running through TEX and then displaying using some DVI tool.

5.6 The Lisp presentation

The Lisp presentation is based on the aspects **Pagename** and **Symbolname**. The Lisp presentation of a page pointed out by a reference r in a web W can be constructed using the definitions below. The Lisp presentation gives some human readable information about the page. One cannot reconstruct a page from the Lisp presentation alone, but in many situations the Lisp presentation will capture all information about a page of interest to the human reader.

The *name* of the page referenced by r (w.r.t. the cache c) is given by

$$n_r = \langle c|r, \mathbf{Codex}, r :: \mathsf{T}, \mathbf{Pagename}\rangle$$

n_r must be a string of small letters (a-z), digits, and hyphens. Furthermore, n_r must contain at least one character and the first character must be a letter. If n_r fails to satisfy these conditions, then the name of the page is the one character string "!". If distinct pages in c have the same name, an "!" is appended to the name (unless the name is "!" in which case no "!" is appended). As an exeption, the name of the page T is the empty string.

The *individual name* of the symbol $s = r :: i$ (w.r.t. the cache c) is given by

$$n_s = \langle c|r, \mathbf{Codex}, s, \mathbf{Symbolname}\rangle$$

n_s must satisfy the same requirements as n_r, or else the symbol name is "!". If distinct symbols on a page have the same name, an "!" is appended to the name (unless the name is "!" in which case no "!" is appended). Predefined symbols, i.e. symbols of form $\mathsf{T} :: b_k$ are treated differently. The predefined symbols listed in Section 4.17 have the names indicated in that section (e.g. $\mathsf{T} :: b_0$ has the name t. The individual name of an unlisted, predefined symbol $\mathsf{T} :: b_k$ is $bd_1 \cdots d_j$ where $d_1 \cdots d_j$ is the decimal representation of k. As an example, the individual name of $\mathsf{T} :: b_{117}$ is "b117".

The *full name* N_s of a symbol $s = r :: i$ (w.r.t. the cache c) is the string $n_r{:}\, n_s$, i.e. the name n_r of the page r appended to a "colon" appended to the individual name n_s of the symbol. For completeness, the full name of T is defined to be "!", so T is the only structure whose full name contains no "colon".

Let ␣ denote a space character. The *S-presentation* of an S-expresion t (w.r.t. the cache c) is thus: If t has form

$$\langle s, t_1, \ldots, t_k \rangle$$

then the S-presentation is

$$(N_s␣T_1␣\ldots␣T_k)$$

where N_s is the full name of the symbol s (w.r.t. the cache c) and T_j is the Lisp presentation of t_j ($j \in \{1, \ldots, k\}$). The S-presentation of T is ().

The *Lisp presentation* of the page referenced by r in a web W is

$$(n␣(b_1␣\cdots␣b_i)␣T)$$

where n is the name of r (w.r.t. the minimal cache that contains r), b_k is the name of the k'th bibliographic reference (w.r.t. the minimal cache that contains r), and T is the S-presentation of the parse tree of the page (w.r.t. the minimal cache that contains r).

- $\langle c|r, \mathbf{Codex}, r :: \mathsf{T}, \mathbf{Pagename}\rangle$.
- $\langle c|r, \mathbf{Codex}, r :: \mathsf{T}, \mathbf{Unpack}\rangle$: Translation to parse tree.
- $\langle c|r, \mathbf{Codex}, s, \mathbf{Symbolname}\rangle$.
- $\langle c|r, \mathbf{Codex}, s, \mathbf{Parameters}\rangle$.
- $\langle c|r, \mathbf{Codex}, s, \mathbf{TeXname}\rangle$.
- $\langle c|r, \mathbf{Codex}, s, \mathbf{Codify}\rangle$: Translaton to codex
- $\langle c|r, \mathbf{Codex}, s, \mathbf{Value}\rangle$: Definition as seen from value function.

5.7 Initial editing of pages

6 Logiweb machines

A *Logiweb machine* is an abstract machine that can execute *Logiweb programs.* Logiweb machines themselves are written in some host programming language.

6.1 The structure of Logiweb machines

A Logiweb machine consists of one program. The program has two logical components, namely the *engine* and the *interface*.

The engine is the component where most CPU-time is spent. The engine is able to execute pure functional programs. More precisely, the engine can reduce combinator terms to root normal form.

The interface is the component that does all the things a pure functional program cannot do such as communicating with the outside world. The interface activates, preempts, and reactivates the engine.

Among other, the interface has a facility for dynamically loading code into the Logiweb machine. Such code can make the engine run more efficiently and can add facilities to the interface. A Logiweb machine is said to be *virgin* from the moment it starts until first time it loads. The virgin Logiweb machine has very few facilities and is very inefficient.

6.2 The boot sequence

When the Logiweb machine is started, the interface reads the command line arguments (argc) and the environment in effect (env). The interface converts argc and env into lists of strings.

The interface uses one particular Logiweb page as its *boot page*. The interface decides on what boot page to use in some, unspecified way. As an example, the interface may use the second element of argc as the reference of the boot page (the first element is the name of the command that started the Logiweb machine).

The interface fetches the boot page and converts the body B of the boot page into a computable function f using the boot representation. We shall refer to that function as the *boot function*.

Then the interface forms a *boot event* x_0 of form

$$x_0 = \langle \mathrm{Boot}, \mathrm{argc}, \mathrm{env}, B \rangle$$

A boot event is an *input event*, i.e. a data structure that represents information that comes to the Logiweb machine from the outside world (c.f. Section 6.6).

Then the interface forms a term x of form

$$x = \langle x_0 \rangle$$

The term x is a one-element list whose sole element is the boot event.

Then the interface commands the Engine to reduce fx. The result of the reduction is a list

$$\langle y_1, y_2, \ldots, y_n \rangle$$

whose elements are *output events* (actually, the full truth is a bit more complicated since the engine merely reduces to root normal form and not necessarily to a list, but the details are deferred).

Each output event represents an action which the interface has to execute. The interface is said to *execute an event* when it executes the action represented by the event.

The interface executes each output event in turn. In principle, the Logiweb machine quits when it has executed all the output events. There are, however, output events that command the interface to reactivate the engine, and this allows the Logiweb machine to enter a loop.

6.3 Output events

The virgin Logiweb machine defines the following output events:

$\langle \mathrm{Stderr}, \mathrm{string} \rangle$	stderr event
$\langle \mathrm{Exec}, \mathrm{process}, \mathrm{handler} \rangle$	exec event
$\langle \mathrm{Extend}, \mathrm{string} \rangle$	extend event

A stderr event commands the interface to write the given string on the users terminal. Under Unix (e.g. Linux), a stderr event more precisely commands the interface to send the given string to the stderr output stream. As an example, if the boot function is

$$\lambda x.\langle\langle \mathrm{Stderr}, \text{"Hello world"}\rangle\rangle$$

then the Logiweb machine will write "Hello world" on stderr and then quit.

Exec and extend events are treated in Section 6.4 and 6.5, respectively.

6.4 Exec events

An exec event

$$\langle \mathrm{Exec}, p, h \rangle$$

contains a process p and a handler h, which are computable functions. An exec event commands the interface to perform the following actions:

First, the interface aborts all events that come after the given exec event. At the time the interface takes action on the exec event, it has already executed the events that occur before the given exec event in the list of output events. For that reason, a list of output events will typically contain at most one exec event and the exec event, if any, will typically occur at the end of the list.

Then the interface commands the engine to reduce p where p was the process of the exec event.

During reduction of p there is an upper bound on the cpu time and memory available. The amount of allocated time and memory is system dependent.

The amount of allocated time may be a measure of the number of miliseconds the cpu spends on reducing p. The amount may also be an amount of real time such that reduction of p is interrupted by a real time clock regardless of the amount of time the cpu has spent on the reduction. The amount may also be a count of reductions, so that the engine counts how many reductions it performs and suspends p when it has performed a certain number of reductions.

The amount of allocated memory may be e.g. all memory available minus a fixed amount of memory that is held in reserve.

If reduction of p is suspended due to time out, it is said to be interrupted by the *watchdog interrupt.* If reduction of p is suspended due to memory shortage, it is said to be interrupted by the *memory interrupt.* If reduction of p succeeds before the watchdog or memory interrupts occur then the reduction is said to be interrupted by the *exit interrupt.*

In any case, reduction of p ends with an interrupt. Each interrupt has an interrupt id which is a Logiweb binary tree. The interrupt ids of the exit, watchdog, and memory interrupts are Exit, Watchdog, and Memory, respectively.

When an interrupt occurs, the interface forms an *interrupt event* x_0 of form

$$x_0 = \langle \text{Interrupt}, \langle i, i_1, \ldots, i_n \rangle, p' \rangle$$

where i is the interrupt id, $i_1, \ldots, i_n$ are parameters of the interrupt (if any) and p' denotes whatever the engine has reduced p to.

Then the interface collects all pending input and forms associated *input events* $x_1, \ldots, x_n$. Each input event represents some input. As an example, if the interface has received a character from stdin, then the list will contain a stdin input event that contains the character. The input events of the virgin Logiweb machine are presented in Section 6.6.

Then the interface constructs a list

$$x = \langle x_0, x_1, \ldots, x_n \rangle$$

where x_0 was the interrupt event and $x_1, \ldots, x_n$ were other input events.

Then the interface commands the engine to reduce hx where h was the handler of the exec event.

From this point on, the Logiweb machine behaves exactly like it did when reducing fx during the boot sequence.

During reduction of hx there is no limit on the time and memory available to the engine (if the engine runs short of memory during reduction of hx, the Logiweb machine prints an error message and quits). Furthermore, the output events generated by hx are sent directly to the interface for execution. In other words, the handler function h has complete control over the Logiweb machine. In contrast, the process p has limited control over the Logiweb machine since it has limited time and memory available and since the output from p is sent to h and not to the interface.

The Logiweb machine is said to be in *supervisor mode* when it reduces the boot function f or a handler h; it is said to be in *user mode* when it reduces a process p.

6.5 Extend events

An extend event commands the interface to *load* the given string. How the string is loaded is system dependent, but the string should typically be expressed in the host language in which the Logiweb machine is implemented. As an example, a Logiweb machine written in C would typically use the C compiler to compile the string and then use the dynamic loader to load the object code into the running Logiweb machine. A Logiweb machine written in Lisp would typically use the Lisp system to load the string.

Extend events allow the Logiweb machine to load software modules into the running Logiweb machine. Such software modules will be refered to as *extensions*. Extensions may define new interfaces, and they may define plugins to the engine that makes the engine more efficient.

When the Logiweb machine needs to load an extension, it will typically need some information about the system on which it is running (e.g. type of hardware, name of host operating system, name of host language, name of compiler). By convention, such information must be present in the boot page.

The Extension interface has to be guarded by anti-virus measures. The simplest and most efficient antivirus measure is to let the boot function load all extensions that have to be loaded and then load an extension that disables the extension interface.

6.6 Input events

The virgin Logiweb machine has the following input events:

$$\begin{array}{l}\langle \mathrm{Boot}, \mathrm{argc}, \mathrm{env}, \mathrm{boot}\rangle \\ \langle \mathrm{Interrupt}, \mathrm{int}, \mathrm{process}\rangle \\ \langle \mathrm{Stdin}, \mathrm{character}\rangle \\ \langle \mathrm{Extended}, \mathrm{bool}, \mathrm{string}\rangle\end{array}$$

The interface forms a boot event during the boot sequence and an interrupt event each time an interrupt occurs.

Whenever the interface executes an exec event, it consults stdin to see if there is any pending input. If there is, the interface reads one character from stdin and forms an input event. The interface reads at most one character from stdin each time it executes an exec event. This prevents the system from being flooded if someone pipes or pastes large amounts of characters into stdin.

When the interface executes an extend event, it immediately loads the given string. The result of loading the string is expressed in an Extended event which contains a Boolean and a string. The Boolean has value T if loading succeeded without errors and has value F otherwise. The string contains a human readable message which can contain whatever the interface thinks might be of interest to someone.

6.7 Restrictions on extensions

By convention, extensions are not allowed to alter the functional behaviour of the engine. Loading an extension may have the effect that the engine can store data in a more efficient format or may have the effect that it can execute certain functions more efficiently. However, any function that the engine can compute after loading should, at least in principle, be computable by the virgin engine.

As an example, an extension is not allowed to add a function whose return value is the time of the day since such a time function is not computable in the virgin engine. If the time of the day is needed, then one has to load an extension which extends the interface such that the interface produces input events that contain the time of the day.

The time function above is an example of an imperative function. Functions that perform input and output are other examples of imperative functions. Imperative functions are not allowed in the engine; they must be placed in the interface and must be made available through input and output events.

As another example, a software module is not allowed to add the parallel "or" function to the engine. Parallel "or" is extensional (as opposed to imperative), but it is nevertheless forbidden in the engine because the virgin engine cannot compute it.

The convention that extensions are not allowed to add previously uncomputable functions to the engine simplifies porting of code and correctness proofs.

An extension may e.g. extend the engine such that it can store integers orders of magnitude more densely than the virgin engine does. An extension may allow the engine e.g. to compute the sum of two integers orders of magnitude faster than the virgin engine. An extension may add e.g. tcp/ip and wallclock interfaces to the interface. These are the intended applications for extensions.

In general, extensions may add any facility to the Logiweb machine, but whatever does not fit into the engine must be put into the interface.

6.8 Authentication

As a special case, extensions may be used for authentication. As an example, consider a public key crypto system in which a public key k is the product of two large primes p and q.

Given k, the virgin engine can, in principle, compute p and q. The owner of the public key (who knows p and q) may extend the engine with a function which can factor any number n, but which returns p and q immediately in the special case where $n = k$.

Such a function speeds up the factoring function in one, special case. In practice, only the engine that belongs to the owner of the public key will be able to factor k. All other engines will require ages to factor k, and that allows the owners engine to do authentication.

6.9 Identification of extensions

By convention, each extension should be placed on a Logiweb page. That Logiweb page will be refered to as the *home page* of the extension.

The typical way of loading an extension would be as follows:

First, the boot function constructs and loads a string which enables access to Logiweb. Then, whenever the program running on the machine needs further extensions, it uses Logiweb messages to fetch the home pages of the extensions. Then a string is extracted from the home page using some function that is built into the program. Finally, that string is loaded using an extend event.

Each input and output event is a list. We shall refer to the first element of an event as the *event identifier*.

Extensions may define new input events, new output events, and new interrupts. These new events and interrupts must be identified by new event and interrupt identifiers. By convention, all such identifiers must have form

$$p :: i$$

where p is the reference of the home page of the extension and i can by any Logiweb binary tree.

By convention, the home page of an extension must contain documentation of the extension in addition to the actual code.

7 The Logiweb protocol

7.1 Purpose

The Logiweb protocol allows a collection of Logiweb nodes to cooperate on maintaining one, consistent Logiweb web. The Logiweb protocol allows nodes to *publish* Logiweb pages, i.e. to make pages available on the web. Furthermore, the protocol allows the nodes to retreive pages from the web and to perform various book keeping.

7.2 Servers and clients

The Logiweb nodes come in two sorts: *servers* and *clients*. Servers are nodes that

- are able to publish pages,
- which conform to the entire Logiweb protocol, and
- which are always up running and always attached to the host networks.

The last requirement is an ideal requirement rather than an absolute requirement. Technological malfunction may prevent a node from running or detach it from the host networks at any time. However, the last requirement says that whoever launches a Logiweb server should try to make it available always.

If a user wants to make a page available on a Logiweb web, then the user must *submit* the page to a server. Once submitted and accepted, the server publishes the page. That a page is published actually means that whenever some Logiweb node asks the server for a copy of the page, the server is willing to send such a copy via the host networks.

Submission, acceptance, and rejection of pages is out of the scope of the Logiweb protocol. Any server must have its own mechanism for submission and its own policy for accepting and rejecting pages. As an example, a server may assign a disk quota for each of its users and may have the policy to publish a page if and only if the submitter stays within that users quota.

Logiweb clients have the following properties:

- They are unable to publish pages.
- They merely conform to part of the Logiweb protocol.
- They don't need to be attached to the host networks permanently.

Logiweb clients can be all sort of programs: browsers, booking systems, software developments systems, editors, proof assistants, or whatever. One may even implement compilers and operating systems as Logiweb clients. The definition of a Logiweb client is essentially negative: a Logiweb client is a program that supports part of the Logiweb protocol, but which does not qualify to be a server (and, hence, is forbidden to submit pages).

7.3 Logiweb messages

As mentioned in Section 2.2, Logiweb nodes exchange Logiweb messages. A Logiweb message consists of a sequence of bytes. Each byte consists of eight bits. The overall format of a Logiweb message is thus:

$$\begin{array}{lcl}
\textit{message} & ::= & \textit{sender receiver cargo} \\
\textit{sender} & ::= & \textit{address} \\
\textit{receiver} & ::= & \textit{address} \\
\textit{cargo} & ::= & \textit{family operation msglength msgbody} \\
\textit{operation} & ::= & \textit{byte} \\
\textit{msglength} & ::= & \textit{cardinal} \\
\textit{msgbody} & ::= & \textit{byte}^* \\
\textit{byte} & ::= & \textit{bit}^8 \\
\textit{cardinal} & ::= & 0\ \textit{bit}^7 \mid 1\ \textit{bit}^7\ \textit{cardinal}
\end{array}$$

A Logiweb/UDP/IPv4 package consists of an IPv4 header, a UDP header, and a Logiweb *cargo* structure. The *sender* and *receiver* part of the Logiweb message must be deduced from the UDP and IPv4 headers. In Logiweb/UDP/IPv4 there is a one-to-one correspondence between Logiweb messages and UDP packages: Each UDP package contains one and only one Logiweb message, and Logiweb messages are never split into more than one UDP package.

A Logiweb/TCP/IPv4 package consists of a *cargo* structure which is sent over a TCP/IPv4 connection. The *sender* and *receiver* part of the Logiweb message must be deduced from the TCP and IPv4 headers. Two Logiweb nodes may exchange several Logiweb message back and forth over a single TCP/IPv4 connection. Logiweb messages are sent back-to-back over the TCP/IPv connection. It is unspecified in the Logiweb protocol when a TCP/IPv4 connection should be closed, and it can be closed by either end, but typical behaviour would be to let the connection stay open until it is closed by the party that created it. In Logiweb/TCP/IPv4 there is no connection between individual TCP/IPv4 packages and Logiweb messages.

A *cardinal* is a finite sequence of bit^7 with some 0 and 1 bits in between which indicate where the cardinal ends. The value of a *cardinal* equals the binary value of the concatenation of all the bit^7 pieces of the cardinal. A cardinal is represented with the least significant bit^7 piece first. As an example, the value of

1000 0000   0000 0011

is $3 \cdot 2^7$.

The value of *cardinal* must be equal to the length of the entire message (including family, operation, length, and body).

Note that host protocols may impose restrictions on the length of a message. As an example, transmission of Logiweb messages using UDP puts restrictions on the length of the messages.

When a message is transmitted using e.g. UDP, then the bytes of the message are transmitted left to right so that the family byte is the first byte transmitted. Each individual byte of the message is transmitted right to left so that the "one" bit of the family byte is the first bit transmitted.

7.4 Challenge and response

Many Logiweb messages contain a *challenge* or a *response* in their cargo. The syntax of challenges and responses reads:

challenge	::=	*scheme cardinal*
response	::=	*scheme cardinal*
scheme	::=	*byte*

If a Logiweb node sends a Logiweb message that contains a *challenge*, then the receiver is expected to send some Logiweb message with an associated *response*. From now on, the sender of a *challenge* will be named the *challenger*, and the sender of the associated *response* will be named the *responder*.

The purpose of sending a *challenge* is to ensure the identity of the responder.

A *challenge* and a *response* consist of a *scheme* and a *cardinal* each. If the *scheme* of a *challenge* is 0000 0000, then the associated valid *response* is a bit string identical to the *challenge*.

If the *scheme* is different from 0000 0000, then the responder applies a responder specific *signature function* to the *challenge* to form a *response*. The *scheme* of the *response* must be equal to the *scheme* of the *challenge* and the *cardinal* of the *response* must be a function of the *scheme* and *cardinal* of the *challenge*. The *response* may also depend on other things such as the various fields of the Logiweb message in which the *response* occurs. To be of any use, the challenger must know something about the signature function of the responder so that the challenger can use the *response* to validate the identity of the responder. How the challenger gets to know something about the signature function is out of the scope of Logiweb.

The default signature function is the identity function, i.e. the function that returns a *response* which is identical to the given *challenge*.

If a challenger knows nothing about the signature function of the responder, then the challenger may still get some confidence in the identity of the responder by using $\mathit{scheme} = 0000\,0000$.

A Logiweb message may contain both a *challenge* and a *response*. This occurs when two parties need mutual confidency in the identity of the other party.

7.5 Get page

When a server receives a $\mathit{getpage}_1$ message, then it typically responds with one or more $\mathit{getpage}_2$ messages. If the server does not store a copy of the requested page, then it responds with a $\mathit{getpage}_3$ message.

The syntax of the *getpage* messages introduced so far reads:

$$\begin{array}{lcl}
\mathit{getpage}_1 & ::= & \mathit{challenge\ index\ length\ reference} \\
\mathit{getpage}_2 & ::= & \mathit{response\ index\ length\ reference\ bit}^* \\
\mathit{getpage}_3 & ::= & \mathit{response\ index\ reference} \\
\mathit{index} & ::= & \mathit{cardinal} \\
\mathit{length} & ::= & \mathit{cardinal}
\end{array}$$

The *response* of $\mathit{getpage}_2$ and $\mathit{getpage}_3$ messages must be valid responses to the *challenge* of the associated $\mathit{getpage}_1$ message.

$\mathit{getpage}_1$ messages may come from clients as well as other servers. The sender of the $\mathit{getpage}_1$ will be referred to as the *requester*.

A $\mathit{getpage}_1$ message contains the reference of a page, an index to a bit within the page, and a length. A $\mathit{getpage}_1$ requests the server to transmit a number of bits of the page, starting with the one pointed out by the index. *length* indicates how many bits the server should transmit.

If the requested range of bits extends beyond the end of the page, then the server does not respond at all. This is a punishment servers use against requesters who ask for information without knowing how much they ask for.

If the *length* of the $\mathit{getpage}_1$ is zero, then the $\mathit{getpage}_1$ message is considered malformed and is discarded by the server.

If the server stores a copy of the requested page, if *length* is positive, and if the requested range of bits fall within the page, then the server either responds with one *getpage*$_2$ message that contains all of the page or with a number of *getpage*$_2$ messages that contain a part each. Each *getpage*$_2$ message contains an index and a length which indicate which page bits the message contains.

If the server responds to a *getpage*$_1$ message by several *getpage*$_2$ messages, then the *response* of each *getpage*$_2$ message is computed individually. If the *scheme* of the *challenge* of the *getpage*$_1$ is 0000 0000, then all responses will be identical to the challenge. Otherwise, the signature function may depend on the contents of the *getpage*$_2$ message so that each *getpage*$_2$ gets its own, individual signature.

7.6 Get page validation

If a server for some reason finds that a *getpage*$_1$ request smells suspicious, it may send a *getpage*$_4$ message to the requester. The requester must respond with a *getpage*$_5$ message within a reasonable time span before the server starts or resumes its transmission. Otherwise, the server will consider the original *getpage*$_1$ request to be an attempt to spam the given requester.

It is out of the scope of the standard to define when a *getpage*$_1$ smells suspicious. Typical behaviour would be to let a server send a *getpage*$_4$ message whenever the requested page is larger than some threshold.

The *getpage*$_4$ and *getpage*$_5$ messages are identical to *getpage*$_1$ messages except that they contain additional challenges and responses. The syntax reads:

$$\begin{array}{lcl} \mathit{getpage}_4 & ::= & \mathit{index\ length\ challenge\ response\ reference} \\ \mathit{getpage}_5 & ::= & \mathit{index\ length\ challenge\ response\ reference} \end{array}$$

The response of the *getpage*$_4$ message must be a valid response to the challenge of the initial *getpage*$_1$ message. The response of the *getpage*$_5$ message must be a valid response to the challenge of the *getpage*$_4$ message. When the server starts or resumes transmission after receiving a valid *getpage*$_5$ message, it does so with *getpage*$_2$ messages that contain valid responses the the challenge in the *getpage*$_5$ message.

A requester can ask for a *getpage*$_4$ be sending a *getpage*$_6$ instead of a *getpage*$_1$, or by sending a *getpage*$_6$ after the server has started its transmission. The syntax reads:

$$\mathit{getpage}_6 \quad ::= \quad \mathit{index\ length\ challenge\ reference}$$

7.7 Reference query

A requester can use *query* messages to find out which pages a server stores. A requester may also use *query* messages to locate a page for which it knows the reference but does not know where to find it.

The body of a Logiweb page can be encrypted, so a server is able to publish pages that are secret in the sense that only nodes that can decrypt them can read their contents.

However, due to the *query* messages, any node can find out exactly which pages a server publishes. Hence, encryption can hide the contents of a page but not the existence and length of the page. Servers are not allowed to keep pages invisible to the *query* system.

Furthermore, any node can read the bibliography of any page. In other words, it is permissible to hide the contents of the body using encryption, but the bibliography cannot be hidden. Anyone who wants to make a secret reference to some other page must include the reference as an encrypted soft link in the message body and must keep the reference out of the bibliography.

The syntax of query messages reads:

$$\begin{array}{lcl}
\mathit{query}_1 & ::= & \mathit{challenge\ reference} \\
\mathit{query}_2 & ::= & \mathit{response\ pagelength\ biblength\ reference} \\
\mathit{query}_3 & ::= & \mathit{response\ timestamp\ prefixlength\ address\ reference} \\
\mathit{query}_4 & ::= & \mathit{response\ timestamp\ prefixlength\ reference} \\
\mathit{pagelength} & ::= & \mathit{cardinal} \\
\mathit{biblength} & ::= & \mathit{cardinal} \\
\mathit{preflength} & ::= & \mathit{cardinal}
\end{array}$$

When a server receives a query_1 message it responds with a query_2, query_3, or query_4 message.

A query_2 message indicates that the server stores a copy of the requested page and indicates the total length of the page and the length of the bibliography of the page. The requester can use the length of the page to allocate buffers and then send one or more $\mathit{getpage}_1$ messages to receive the page. The requester can use *biblength* if it merely wants to receive the bibliography. The latter allows a client to locate all descendants of a page before it starts requesting the page and its descendants.

A query_3 message indicates that the server stores no copy of the requested page but knows whom to ask next. The *address* field of the query_3 message points out a server which is supposed to know more about the given page than the present server.

A query_4 message indicates that the server stores no copy of the requested page and does not know whom to ask for more information.

A query_3 and query_4 message contains a *prefixlength* whose value is a number n. The value n indicates that the server stores a page whose reference is identical to the requested reference up to but excluding bit n of the references (bit 0 is the first bit of the reference). The *prefixlength* is the largest number n for which that is true. As an example, suppose a server stores two pages, one with address $\langle 0,0,0\rangle$ and one with address $\langle 0,1,0,0\rangle$. If that server receives a query_1 for a page with reference $\langle 0,1,1,0\rangle$, then it responds with a query_3 or query_4 where *preflength* is 2.

A requester may efficiently find all pages published by a server by a sequence of query_1. However, a server may publish or withdraw pages over time, and for that reason query_3 and query_4 messages contain a *timestamp* which indicates the time at which the given information was valid.

If a requester R sends a query_1 message to a server S, then S is merely allowed to point out another server S' using a query_3 message if S' can respond to the query_1 message either by a query_2 message or by a query_3 message with a larger *prefixlength*.

If a server receives a $query_1$ message for a page it does not store, then the server could pretend that it stored the page by fetching the page from some other server and then answering that it did store the page. That, however, is forbidden. However, if a server receives a $getpage_1$ message for a page it does not store, then the server is allowed but not required to fetch the page from some other server and pass it on to the requester. Hence, servers are required to give immediate responses to $query_1$ messages which honestly reflect the situation at the moment the message is received.

7.8 Complaints

If a requester tries to locate a page, it will typically send a $query_1$ message to some server which responds with a message M_1. If M_1 is a $query_3$, then the requester will send a $query_1$ message to the server pointed out in M_1 which responds with a message M_2 and so on. The process can stop in four ways:

1. After a while, the requester receives a $query_2$ message in which case the search was successful.
2. After a while, the requester receives a $query_4$ message in which case the search was unsuccessful.
3. After a while, the requester receives a $query_3$ message whose *prefixlength* is less or equal the *prefixlength* of the previous $query_3$ message. The search is considered unsuccessful.
4. After a while, the requester hits a non-responding server in which case the search was unsuccessful.

If a requester receives one or more $query_3$ messages but the search ends without success, then all the $query_3$ messages are considered misleading. The requester should inform the servers that they give misleading answers. The requester does so by sending a $query_5$ message. The syntax reads:

$$query_5 \quad ::= \quad timestamp\ reference$$

The $query_5$ is a complaint that says that the requester was dissatisfied with the answer it got. The *timestamp* must be a copy of the *timestamp* of the $query_3$ which the requestor complains about. It is outside the scope of Logiweb to specify how servers should react to $query_5$ messages.

If a requester performs and unsuccessful search but then manages to locate the page anyway, then the requester may tell where it found the page using a $query_6$ message:

$$query_6 \quad ::= \quad timestamp\ address\ reference$$

As can be seen, any client must know the address of at least one server to get started. It is outside the scope of Logiweb to define how a client finds the first server, but since servers are required to be always on, the programmer of a client may e.g. decide to place the Logiweb address of a server in an environment variable. In this case, the Logiweb address could be expressed in some, external format such as “UDP/IPv4:logic.dk:117”, where the domain name can be translated into an IP address using DNS.

7.9 Summary

The table below summarises the syntax of all Logiweb messages and gives the operation code “op” of each message type.

name	op	syntax
$getpage_1$	0	*challenge index length reference*
$getpage_2$	1	*response index length reference bit**
$getpage_3$	2	*response index reference*
$getpage_4$	3	*index length challenge response reference*
$getpage_5$	4	*index length challenge response reference*
$getpage_6$	5	*index length challenge reference*
$query_1$	6	*challenge reference*
$query_2$	7	*response pagelength biblength reference*
$query_3$	8	*response timestamp prefixlength address reference*
$query_4$	9	*response timestamp prefixlength reference*
$query_5$	10	*timestamp reference*
$query_6$	11	*timestamp address reference*

8 A Logiweb machine implementation

8.1 Cells

The vast majority of data structures in the Logiweb machine are kept in the *heap*. The heap is a collection of *cells*. Each cell contains four *cell pointers* (i.e. pointers to cells). The four cell pointers are named *root*, *head*, *tail*, and *link*, respectively. Cells are of of type *cell* and cell pointers are of type *pnt*. The declarations of the types read:

```
typedef struct cellstruct cell;
typedef cell *pnt;
struct cellstruct{pnt root,head,tail,link;};
```

To begin with, the heap consists of a single array of cells:

```
cell heap[INIT_HEAP_SIZE];
```

Using the extension interface, the Logiweb machine may allocate more memory for the heap. In that case, the heap will no longer consist of a single array of cells but will consist of a collection of arrays. Each array, i.e. each consecutive area of cells will be referred to as a *chunk*.

8.2 Scalar types

The virgin Logiweb machine works with three scalar types: cell pointers, *dynamic functions*, and integers. Whenever the Logiweb machine stores a scalar in a cell, it casts the scalar to be a cell pointer. Hence, even though a cell formally contains four cell pointers, it may actually contain a mixture of cell pointers, dynamic functions, and integers.

Dynamic functions are pointers to functions that take no arguments and produce no return value. Dynamic functions are of type *fct*:

```
typedef void (*fct)(void);
```

It is important for the implementation that an integer or a dynamic function can be cast to a cell pointer and back without loss of information. In other words, cell pointers must take up at least as much memory as integers and dynamic functions. Furthermore, it is important that integers can be large enough to count all cells twice. In other words, if the largest integer is N then the heap is not allowed to contain more than $N/2$ cells.

8.3 Organisation of the heap

The Logiweb machine stores a number of cell pointers in global variables. These global variables are accessible to all code of the Logiweb machine, including code that is loaded using the extension interface. All access to the heap goes through these global variables. In particular, dynamic functions read from and write to the global variables.

The heap is said to be *consistent* whenever the cells and global variables are organised as described in the following. Whenever the heap is consistent, it is possible to tell the type of every field of every cell if one has access to all global variables and all cells.

The heap is required to be consistent at various points in the code of the Logiweb machine. These points will be referred to as *check points*.

Among other, the heap must be consistent whenever the Logiweb machine calls a dynamic function, and the heap must be consistent whenever a dynamic function returns. Extensions may introduce new dynamic functions, and it is the responsibility of the programmer of the extension to ensure that each new dynamic function maintains consistency.

It is very difficult to write functions that maintain consistency, but violation of consistency has grave consequences. If one is very lucky, the Logiweb machine produces a core dump, but the machine may also begin to behave strangely at some point in time long after the consistency was violated. For that reason, one should think more than twice before writing an extension, and one should think more than twice before using an extension written by others.

For the sake of test and debugging, the Logiweb machine contains conditionally compiled code that can check the heap for consistency at all or at some checkpoints.

8.4 Reference counts

All global variables and all cell fields, are classified as *counted* or *uncounted* as follows:

- Global variables are classified as uncounted.
- Root and link fields are classified as uncounted.
- The head field of a cell is classified as counted if and only if the least significant bit of the root is non-zero.
- The tail field of a cell is classified as counted if and only if the second least significant bit of the root is non-zero.

We shall refer to the least and second least bits of the root of a cell as the *mode bits* of the cell.

A cell is referred to as a *counting cell* if there is at least one counted field that points to it.

The link field of a counting cell contains an integer. This integer will be referred to as the *reference count* of the cell.

The reference count of each counting cell must be equal to the number of counted pointers that point to the cell. As can be deduced from the statements above, the reference count of any counting cell is positive. The reference count cannot be zero since, by definition, a cell is counting if at least one counting pointer points to it.

8.5 Linked lists

A cell is referred to as a *linked cell* if it is not counting. The contents of the link field of a linked cell will be referred to as the *link pointer* of the cell.

At each check point, every link pointer must be null or point to a linked cell, no two link pointers are allowed to point to the same cell, and the link pointers must form an acyclic graph.

The consistency rules above ensure that the linked cells form a number of disjoint lists, each of which has a start and an end. We shall refer to these lists as the *linked lists* of the heap. As a special case, a linked list may have only one element in which case that element is both the start and the end of the list.

As can be deduced from the rules above, all pointers to linked cells are uncounted since if they were counted, the cell would be counting.

8.6 Root functions

As mentioned, the mode bits of each cell determine whether or not the head and tail fields are counted. The remaining bits of the root constitute a dynamic function which we shall refer to as the *root function* of the cell. To obtain the root function of a cell one must clear the two least significant bits of the root and then cast the result to a dynamic function. If *p* points to a cell, then one may get the root function of the cell thus:

```
(fct)(p->root&~3)
```

Executing the dynamic function may be done thus:

```
(fct)(p->root&~3)();
```

As a special case, a root function may be null in which case it does not make sense to execute it. The root function of counted cells are never null.

8.7 The chunk list

As mentioned, the heap consists of a number of chunks where each chunk is a consecutive area of cells. The first cell in each chunk will be referred to as a *chunk cell*. Chunk cells are linked cells.

At each check point, all chunk cells must form a single, linked list. The global variable *chunk* must point to the start of this chunk list.

The root of each chunk cell must be zero. The head of each chunk cell must point to the first memory location after the chunk. Note that the head of each chunk cell is uncounted since the root is zero. The tail of each chunk cell must be zero.

The chunk list is merely used for consistency checks. It serves no purpose in production versions of the virgin Logiweb machine, but extensions may depend on the presence of chunk cells.

8.8 The free and reserved lists

The global variables *free* and *reserved* point to linked lists of cells. We shall refer to those lists as the *free list* and *reserved list*, respectively.

Whenever the Logiweb machine needs to allocate a new cell, it uses the cell that “free” points to and moves “free” to the next cell in the list.

If the Logiweb machine is in supervisor mode and “free” is NULL when the Logiweb machine tries to allocate a cell, then the Logiweb machine writes an error message to stderr and quits.

If the Logiweb machine is in user mode and “free” is NULL when the Logiweb machine tries to allocate a cell, then the Logiweb machine sets “free” to the value of “reserved” and sets “reserved” to NULL. Then the Logiweb machine clears the stack as described in Section 8.9 and then it performs a memory interrupt as described in Section 6.4.

Whenever the Logiweb machine takes a cell from the free list, it looks at the mode bits of the allocated cell. If these bits indicate that the head or tail fields contain counted pointers, then the Logiweb machine decrements the reference count of the cells that these fields point to. If a reference count of a cell reaches zero, then that cell is put on the free list. Hence, in some cases, allocation of a cell makes the length of the free list increase.

8.9 The stack

The two global variables *top* and *bottom* point to the start and end of a linked list. We shall refer to this list as the *stack*.

The stack is manipulated by dynamic functions. When an interrupt occurs, the Logiweb machine sets the link field of the end of the stack to the value of "free" and then sets "free" to the value of "top". Effectively, all of the stack is deallocated in a short, fixed amount of time. This allows the machine has to respond immediately to interrupts, which is important e.g. if some extension makes the machine handle hardware interrupts.

Deallocation of the entire stack in case of interrupts has the drawback that the machine has to spend time on rebuilding the stack when a process is resumed. It is a design choice to accept this drawback. The choice is motivated by two benefits: Firstly, interrupts can be handled immediately. Secondly, having stacks of stopped processes hanging around makes it dangerous to let handlers share memory with user processes.

8.10 The superlist

The virgin Logiweb machine has four linked lists that are described until further: the chunk list, the free list, the reserved list, and the stack. Extensions may add to the number of linked lists. To allow the consistency check code to access all lists, the *superlist* is a list that refers to all linked lists except the chunk list and the superlist itself. Furthermore, the superlist may refer to counted cells. The latter allows counted cells to exist even if they are not referred to from any other linked list.

The global variable *super* refers to the start of the superlist. The elements of the superlist have the following format: The head of the element points to the start of a linked list or to a counted cell. The root of the element is zero if the head points to a linked list and one otherwise. The tail of the element points to the previous element of the superlist so that the superlist is a doubly linked list. The tail of the first element of the superlist is null.

8.11 Representation of terms

The Logiweb machine constantly has to reduce terms. The term to be reduced can be the boot function f applied to the init message, it can be a process p, and it can be a handler h applied to input messages.

Terms to be reduced are represented by structures of counted cells. Such counted cells will be referred to as *nodes*.

In the virgin Logiweb machine, the root field of nodes can have eleven possible values which will be referred to under the following names (c.f. Section 4.24):

S	S_1	S_2
K	K_1	
P	P_1	P_2
T	I	A

The eleven entities above all denote values of type pnt which contain two mode bits and a function pointer.

The mode bits of T, S, K, and P are both zero, so nodes with these values in the root field contain no counted pointers in the head and tail fields. Nodes which these values in the root field must contain zeros in the head and tail fields.

The mode bits of A, S_2, and P_2 are both non-zero, so nodes with these values in the root field contain counted pointers in both the head and the tail fields.

The mode bits of I, S_1, K_1, and P_1 indicate that nodes with these values in the root field contain contain a counted pointer in the head field and no counted pointer in the tail field. Nodes which these values in the root field must contain a zero in the tail field.

A node with S in the root field and zeros in the head and tail fields will be denoted

$$(\mathsf{S})$$

A node with S_1 in the root field, x in the head field, and zero in the tail field will be denoted

$$(\mathsf{S}_1 x)$$

A node with S_2 in the root field and x and y in the head and tail fields, respectively, will be denoted

$$(\mathsf{S}_2 xy)$$

Nodes with other of the eleven possible root values are treated similarly.

Nodes represent terms as follows:

Node	Term
(S)	S
$(\mathsf{S}_1 x)$	$\mathsf{S}x$
$(\mathsf{S}_2 xy)$	$\mathsf{S}xy$
(K)	K
$(\mathsf{K}_1 x)$	$\mathsf{K}x$
(P)	P
$(\mathsf{P}_1 x)$	$\mathsf{P}x$
$(\mathsf{P}_2 xy)$	$\mathsf{P}xy$
(T)	T
$(\mathsf{I}x)$	x
$(\mathsf{A}xy)$	xy

8.12 Reduction rules

The virgin Logiweb machine uses the following reduction rules to reduce terms:

$$\begin{array}{lcl}
(\mathsf{A}(\mathsf{S})u) & \to & (\mathsf{S}_1 u) \\
(\mathsf{A}(\mathsf{S}_1 x)u) & \to & (\mathsf{S}_2 xu) \\
(\mathsf{A}(\mathsf{S}_2 xy)u) & \to & (\mathsf{A}(\mathsf{A}xu)(\mathsf{A}yu)) \\
(\mathsf{A}(\mathsf{K})u) & \to & (\mathsf{K}_1 u) \\
(\mathsf{A}(\mathsf{K}_1 x)u) & \to & (\mathsf{I}x) \\
(\mathsf{A}(\mathsf{P})u) & \to & (\mathsf{P}_1 u) \\
(\mathsf{A}(\mathsf{P}_1 x)u) & \to & (\mathsf{P}_2 xu) \\
(\mathsf{A}(\mathsf{P}_2 xy)u) & \to & \cdots \\
(\mathsf{A}(\mathsf{T})u) & \to & (\mathsf{IT}) \\
(\mathsf{A}(\mathsf{I}x)u) & \to & (\mathsf{A}xu) \\
(\mathsf{A}(\mathsf{A}xy)u) & \to & \cdots
\end{array}$$

The Logiweb machine reduces $(\mathsf{A}(\mathsf{A}xy)u)$ by reducing $(\mathsf{A}xy)$ to some form r that does not have A in the root and then reducing $(\mathsf{A}ru)$. The Logiweb machine reduces $(\mathsf{A}(\mathsf{P}_2 xy)u)$ by reducing u to some form r that does not have A in the root and then reducing $(\mathsf{A}(\mathsf{P}_2 xy)r)$ according to the following rules:

$$\begin{array}{lcl}
(\mathsf{A}(\mathsf{P}_2 xy)(\mathsf{S})) & \to & (\mathsf{I}y) \\
(\mathsf{A}(\mathsf{P}_2 xy)(\mathsf{S}_1 s)) & \to & (\mathsf{I}y) \\
(\mathsf{A}(\mathsf{P}_2 xy)(\mathsf{S}_2 st)) & \to & (\mathsf{I}y) \\
(\mathsf{A}(\mathsf{P}_2 xy)(\mathsf{K})) & \to & (\mathsf{I}y) \\
(\mathsf{A}(\mathsf{P}_2 xy)(\mathsf{K}_1 s)) & \to & (\mathsf{I}y) \\
(\mathsf{A}(\mathsf{P}_2 xy)(\mathsf{P})) & \to & (\mathsf{I}y) \\
(\mathsf{A}(\mathsf{P}_2 xy)(\mathsf{P}_1 s)) & \to & (\mathsf{I}y) \\
(\mathsf{A}(\mathsf{P}_2 xy)(\mathsf{P}_2 st)) & \to & (\mathsf{I}y) \\
(\mathsf{A}(\mathsf{P}_2 xy)(\mathsf{T})) & \to & (\mathsf{I}x) \\
(\mathsf{A}(\mathsf{P}_2 xy)(\mathsf{I}s)) & \to & (\mathsf{A}(\mathsf{P}_2 xy)s)
\end{array}$$

In addition to the rules above, the Logiweb machine uses the following reduction rule whenever possible:

$$(\mathsf{I}(\mathsf{I}z)) \to (\mathsf{I}z)$$

More precicely, whenever the Logiweb machine forms a node of form $(\mathsf{I}x)$ it performs the reduction above if x is of form $(\mathsf{I}z)$. For that reason, the head of an I node never points at an I node. The consistency check code tests this.

8.13 Reduction algorithm

The Logiweb machine reduces a term t thus: It sets the global variable *redex* to point at t. If t neither has form $(\mathsf{A}xy)$ nor $(\mathsf{I}x)$ then reduction halts because the term is already on normal form. If t has form $(\mathsf{I}x)$ then the Logiweb machine sets "redex" to point at x and starts over.

If t has form $\mathsf{A}xy$ then the Logiweb machine allocates a cell from the heap and sets "stack" to point at the cell. The head, tail, and link fields of the cell are set to zero and the root is set to point at a function to be executed when reduction ends.

```
(fct)(redex->head->root&~3)();
```

9 A server implementation

This chapter gives the details of one particular server implementation. The server is written in C and runs under Linux.

9.1 Name

lgws - Logiweb server

9.2 Synopsis

lgws LGW ADM WD

9.3 Description

Starts up a Logiweb server.

LGW and ADM are udp/ip port numbers and WD is the name of a working directory. The server uses LGW for the Logiweb protocol and uses ADM for the *administration protocol* which is described later.

9.4 Page submission

To submit a page, a user must store the contents of the page in a file and place the file somewhere under the working directory of the server. Then the user must notify the server via the administration protocol. The notification must contain the name of the file (relative to the working directory of the server).

The contents of the file must have the format

> *contents padbits*

where *padbits* denotes from zero to seven "zero" bits which ensure that the total length of the file is a multiple of eight bits.

When the server is notified about a submission, it assigns a reference to the page. To do so it assigns a time stamp and computes a key. Then the server writes the reference and the file name to a file named "assoc" in the working directory. The assoc file thus defines the association between references and file names. Then the server publishes the page, i.e. it makes it available to the outside world via the Logiweb protocol.

The purpose of the assoc file is to allow the server to restart after a system crash.

The user id of the assoc file equals the user id of the server. The assoc file has protection `-rw-r--r--`.

9.5 Administration protocol

The administration protocol has a facility for translating between file names and Logiweb references. The Logiweb messages for translating between file names and Logiweb references are as follows:

getref_1	::=	*name*
getref_2	::=	*reference*
getref_3	::=	*name*
$\mathit{getname}_1$	::=	*reference*
$\mathit{getname}_2$	::=	*name*
$\mathit{getname}_3$	::=	*reference*
name	::=	$\mathit{length}\ \mathit{bit}^*$

When the server receives a getref_1 message, it

A user may request translation from a file name to a Logiweb reference by sending a getref_1 message to the server. The getref_1 message contains the name to be translated.

When a server receives a getref_1 message, it responds with a getref_2 message if it stores a Logiweb page with that local name and is willing to translate the name. Otherwise, it responds with a getref_3 message.

A user may request translation from a reference to a local name by sending a $\mathit{getname}_1$ message to the server. The $\mathit{getname}_1$ message contains the reference to be translated.

When a server receives a $\mathit{getname}_1$ message, it responds with a $\mathit{getname}_2$ message if it stores a Logiweb page with that reference and is willing to translate the reference into a local name. Otherwise, it responds with a $\mathit{getname}_3$ message.

Even though it is legal to let a server assign several local names to a single Logiweb page, the Logiweb protocol merely allows to ask for one of them. If a Logiweb page has several local names, it is up to the server to choose which one it will translate a given reference into.

getref and *getname*

getref_1	0	*name*
getref_2	1	*reference*
getref_3	2	*name*
$\mathit{getname}_1$	3	*reference*
$\mathit{getname}_2$	4	*name*
$\mathit{getname}_3$	5	*reference*

10 Index