ONCE UPON AN ALGORITHM

HOW STORIES EXPLAIN COMPUTING

BY MARTIN ERWIG

Contents

Table 1	5
Figure 1	6
Figure 2	7
Diagram 1	8
Diagram 2	9
Diagram 3	10
Figure 1.1	11
Table 1.1	12
Figure 1.2	13
Figure 1.3	14
Figure 2.1	15
Diagram 4	16
Diagram 5	17
Diagram 6	18
Diagram 7	19
Diagram 8	20
Diagram 9	21
Diagram 10	22
Diagram 11	23
Figure 3.1	24
Diagram 12	25
Diagram 13	26
Diagram 14	27
Figure 4.1	28
Diagram 15	29
Diagram 16	30
Diagram 17	31
Diagram 18	32
Figure 5.1	33
Diagram 19	34
Diagram 20	35
Diagram 21	36
Diagram 22	37
Diagram 23	38

Diagram 24	39
Figure 5.2	40
Figure 6.1	41
Figure 6.2	42
Figure 6.3	43
Figure 6.4	44
Figure 6.5	45
Diagram 25	46
Table 7.1	47
Diagram 26	48
Figure 8.1	49
Diagram 27	50
Table 8.1	51
Figure 8.2	52
Diagram 28	53
Diagram 29	54
Diagram 30	55
Diagram 31	56
Diagram 32	57
Figure 8.3	58
Figure 8.4	59
Diagram 32b	60
Figure 9.1	61
Diagram 33	62
Diagram 34	63
Figure 9.2	64
Figure 10.1	65
Figure 10.2	66
Figure 11.1	67
Figure 11.2	68
Figure 11.3	69
Figure 11.4	70
Diagram 35	71
Figure 12.1	72
Figure 13.1	73
Diagram 36	74

Diagram 37	75
Diagram 38	76
Diagram 39	77
Diagram 40	78
Figure 13.2	79
Diagram 41	80
Figure 13.3	81
Diagram 42	82
Figure 13.4	83
Diagram 42b	84
Figure 15.1	85
Figure 15.2	86
Diagram 43	87
Figure 15.3	88
Figure 15.4	89
Figure 15.5	90

Table 1

Story	Chapters	Topics
Part I		
Hansel and Gretel	1, 2	Computation and Algorithms
Sherlock Holmes	3, 4	Representation and Data Structures
Indiana Jones	5, 6, 7	Problem Solving and Its Limitations
Part II		
Over the Rainbow	8, 9	Language and Meaning
Groundhog Day	10, 11	Control Structures and Loops
Back to the Future	12, 13	Recursion
Harry Potter	14, 15	Types and Abstraction

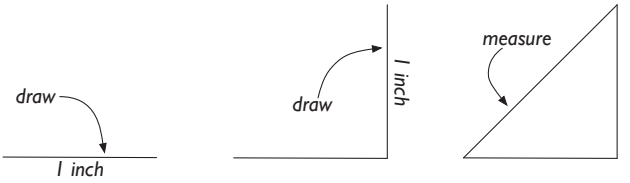


Figure 1 Computing the square root of 2 using a pencil and a ruler.

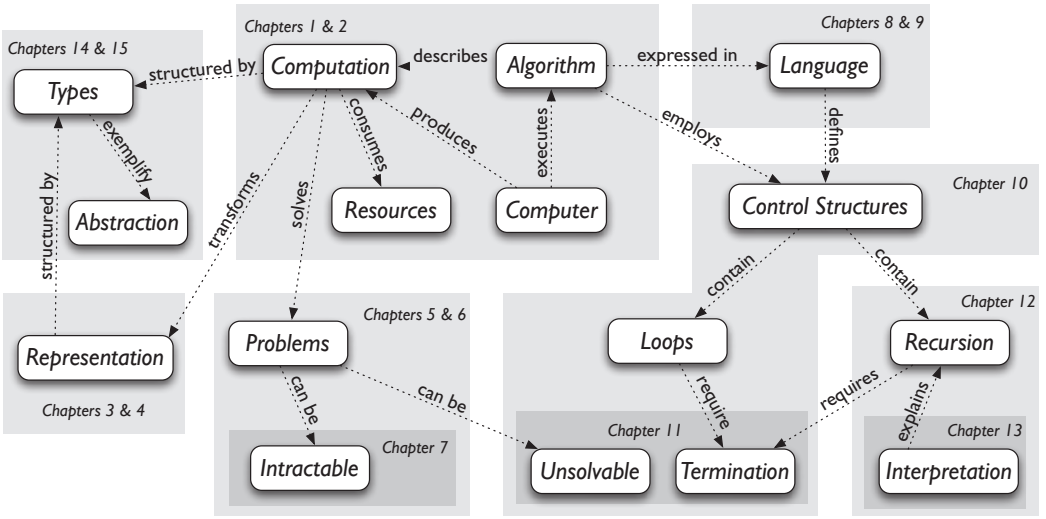


Figure 2 Concepts of computation and how they are related.

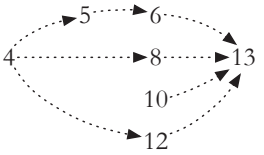


Diagram 1

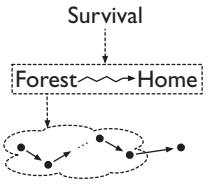


Diagram 2

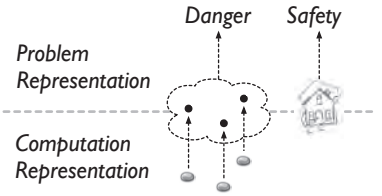

Danger
Safety
Problem
Representation
Computation
Representation

Diagram 3

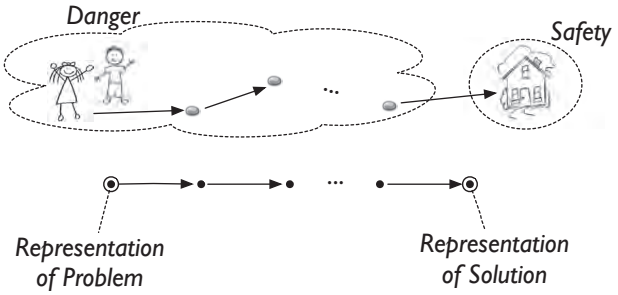


Figure 1.1 Computation is a process for solving a particular problem. Usually, a computation consists of several steps. Starting with a representation of the problem, each step transforms the representation until the solution is obtained. Hansel and Gretel solve the problem of surviving through a process of changing their position step-by-step and pebble-by-pebble from within the forest to their home.

Table 1.1

Computation Representation		*Problem Representation*	
Object	**Represents**	**Concept**	**Represents**
One pebble	Location in forest	Location in forest	Danger
	Home	Home	Safety
All pebbles	Path out of forest	Path out of forest	Problem Solution

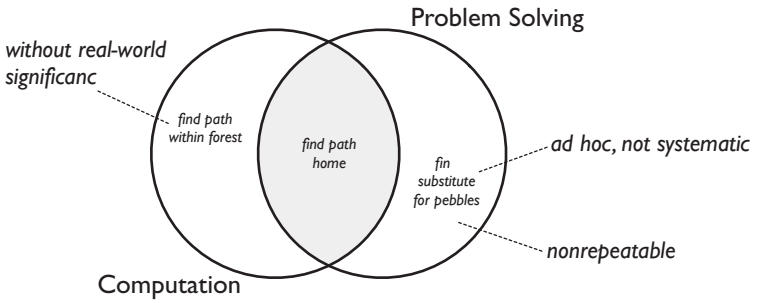


Figure 1.2 Distinguishing problem solving from computation. A computation whose effect carries no meaning in the real world does not solve any problems. An ad hoc solution to a problem that is not repeatable is not a computation.

Figure 1.3 A path that illustrates a possible dead end in an algorithm. *Left:* Visiting the pebbles in the reverse order leads to Hansel and Gretel's home. *Right:* Since pebbles B, C, and D are all within viewing distance from one another, Hansel and Gretel could choose to go from D to B and then to C. At that point, however, they are stuck because no pebble that they haven't visited before is visible from C. Specifically, they cannot reach A, which is the next pebble on the path home.

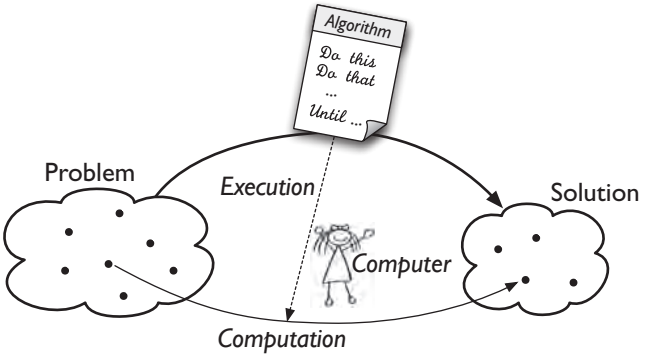


Figure 2.1 The execution of an algorithm generates a computation. The algorithm describes a method that works for a whole class of problems, and an execution operates on the representation of a particular example problem. The execution must be performed by a computer, for example, a person or a machine, that can understand the language in which the algorithm is given.

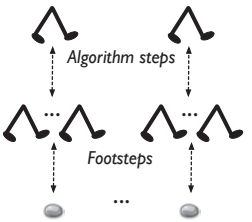


Diagram 4

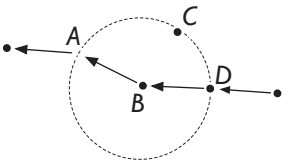


Diagram 5

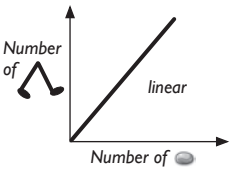


Diagram 6

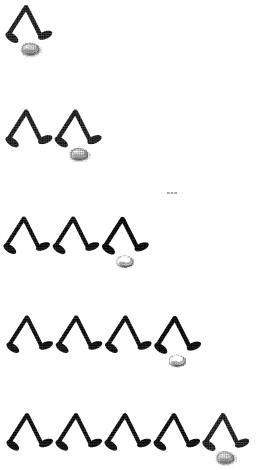


Diagram 7

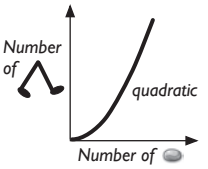


Diagram 8

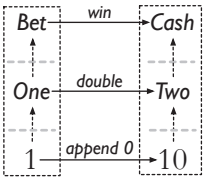


Diagram 9

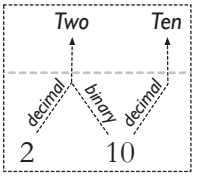


Diagram 10

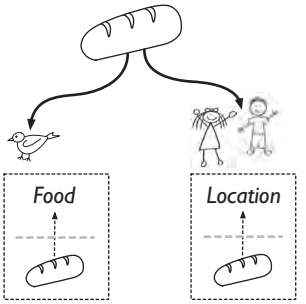


Diagram 11

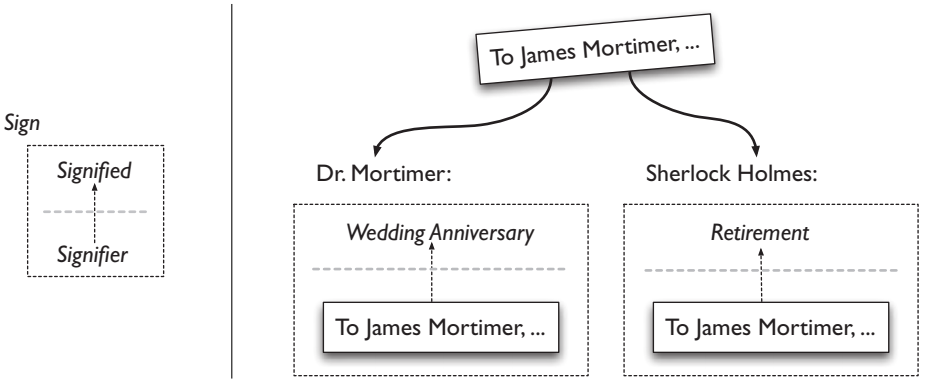


Figure 3.1 Signs are the basis for representation. A sign consists of a signifier that represents some concept, called the signified. One signifier can stand for different concepts.

Diagram 12

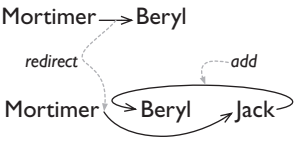


Diagram 13

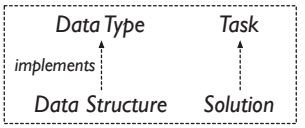


Diagram 14

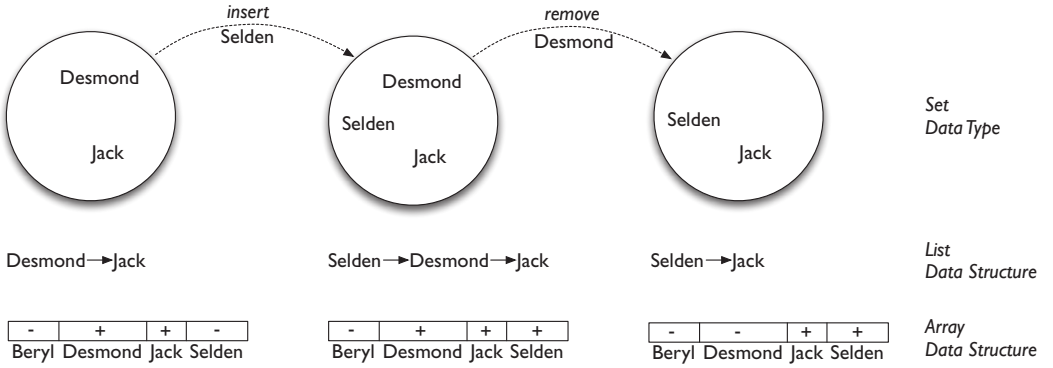


Figure 4.1 A data type can be implemented by different data structures. Inserting an element into a list can be done by simply adding it to the front of the list, but removal requires traversing the list to find it. With an array, insertion and removal are done by directly accessing the array cell indexed by an element and changing the mark accordingly. Arrays allow faster implementation, but lists are more space efficient.

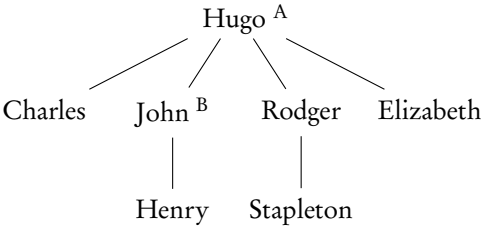


Diagram 15

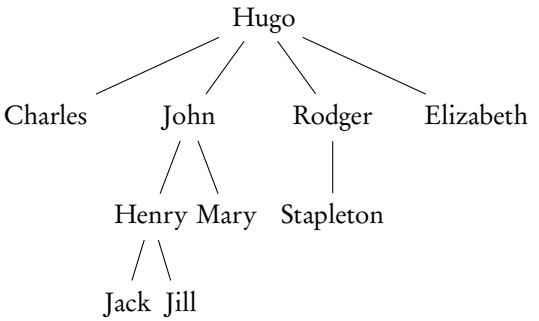

Hugo
Charles
John
Rodger
Elizabeth
Henry
Mary
Stapleton
Jack
Jill

Diagram 16

Diagram 17

G:1 → o:1 → d:1

G:1 → o:2 → d:1 → I:1 → e:1 → h:1 → v:1 → a:1

G:1 → o:4 → d:1 → I:1 → e:4 → h:4 → v:3 → a:4 → J:1 → Y:2 → w:1

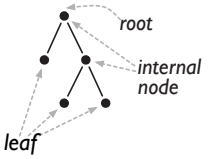


Diagram 18

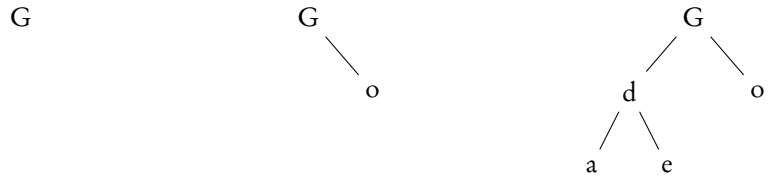


Figure 5.1 Three example binary trees. *Left:* A tree of only one node. *Middle:* A tree whose root has a right subtree of a single node. *Right:* A tree whose root has two subtrees. All three trees have the binary search property, which says that nodes in left subtrees are smaller than the root, and nodes in right subtrees are larger than the root.

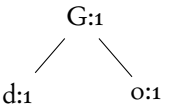
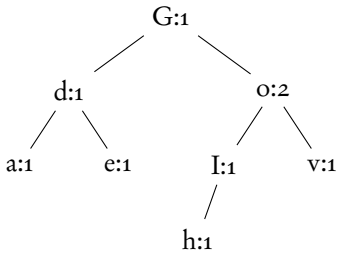
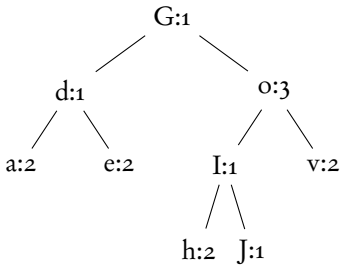


Diagram 19



After adding *Iehova*



After adding *Jehova*

Diagram 20

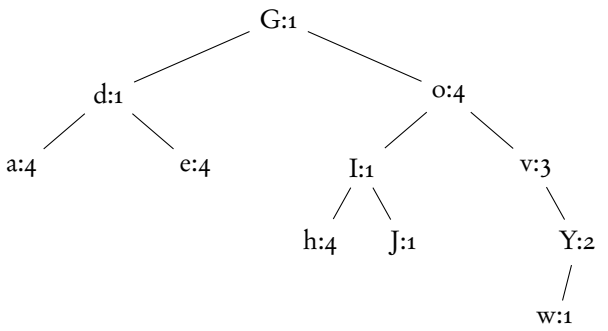


Diagram 21

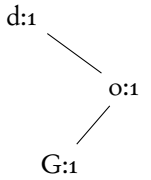


Diagram 22

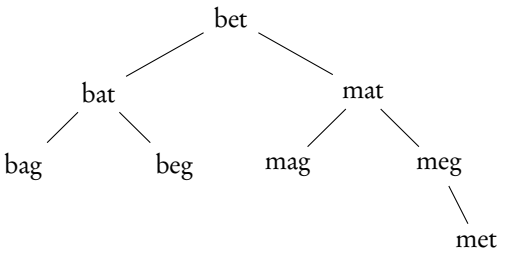


Diagram 23

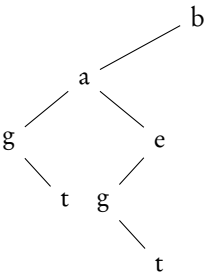


Diagram 24

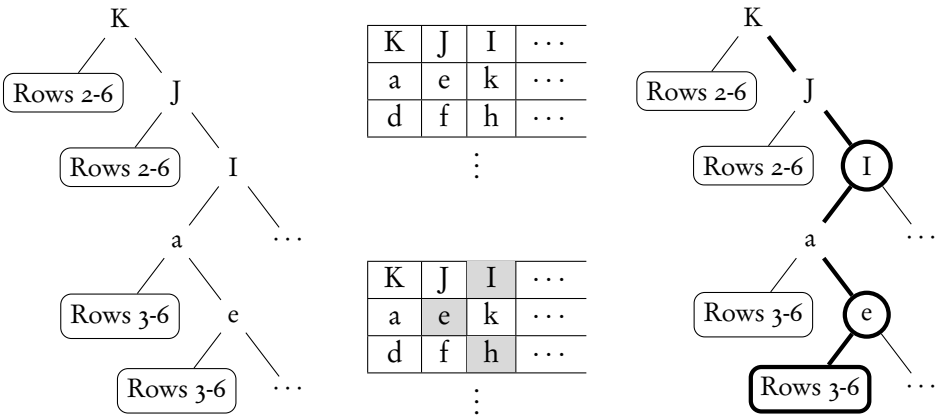


Figure 5.2 A trie data structure and how to search it. *Left:* A trie representation where left subtrees represent possible word continuations of the letter in the parent node (represented by the rounded rectangles) and right subtrees represent alternatives to the parent node. *Middle top:* The tile floor representation of the trie on the left in which the common subtrees are shared through single tile rows. *Middle bottom:* Three selected tiles that spell the beginning of the word *Iehova*. *Right:* A path through the trie marked by bold edges for the selected tiles. The circled nodes correspond to selected tiles.

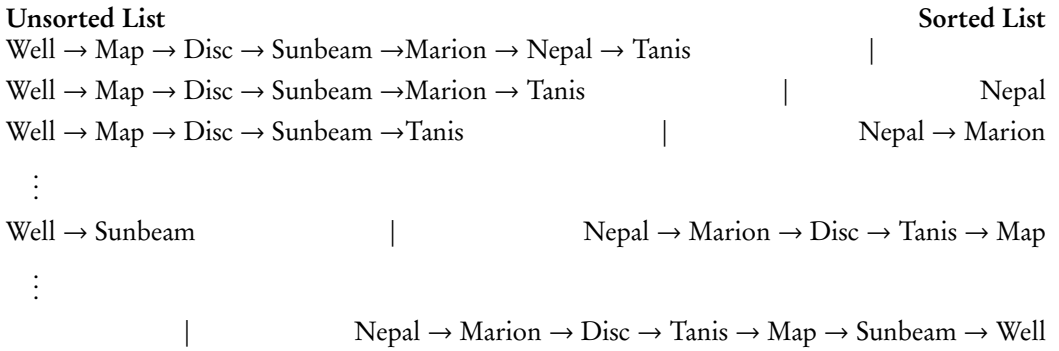


Figure 6.1 Selection sort repeatedly finds the smallest element in an unsorted list (*left of vertical bar*) and appends it to the sorted list (*right*). Elements are not sorted by name but according to the dependency of tasks they represent.

Unsorted List	**Sorted List**
Well → Map → Disc → Sunbeam →Marion → Nepal → Tanis	
Map → Disc → Sunbeam →Marion → Nepal → Tanis	Well
⋮	
Sunbeam → Marion → Nepal → Tanis	Disc → Map → Well
Marion → Nepal → Tanis	Disc → Map → Sunbeam → Well
⋮	
	Nepal → Marion → Disc → Tanis → Map → Sunbeam → Well

Figure 6.2 Insertion sort repeatedly inserts the next element from the unsorted list (*left of the vertical bar*) into the sorted list (*right*).

Well → Map → Disc → Sunbeam →Marion → Nepal → Tanis

Disc → Marion → Nepal | Tanis | Well → Map → Sunbeam

Nepal → Marion → Disc | Tanis | Map → Sunbeam → Well

Nepal → Marion → Disc → Tanis → Map → Sunbeam → Well

Figure 6.3 Quicksort splits a list into two sublists of elements that are smaller and larger, respectively, than a chosen pivot element. Then those two lists are sorted, and the results are appended together with the pivot element to form the resulting sorted list.

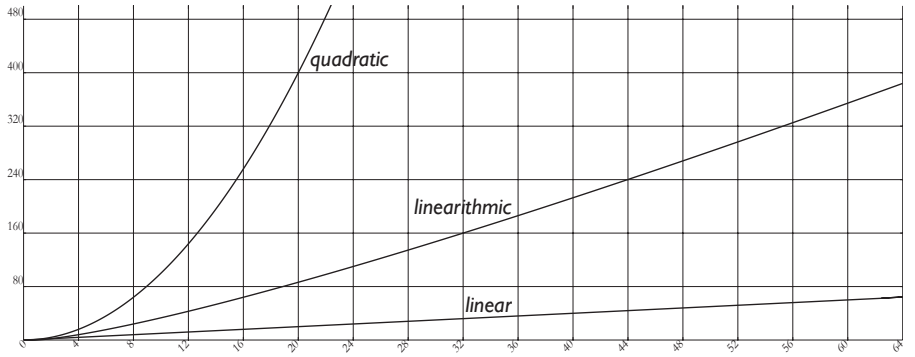


Figure 6.4 Comparison of linearithmic, linear, and quadratic runtimes.

[1] Well → Map → Disc → Sunbeam →Marion → Nepal → Tanis

[2] Well → Map → Disc → Sunbeam | Marion → Nepal → Tanis

[3] (Well → Map | Disc → Sunbeam) | (Marion → Nepal | Tanis)

[4] ((Well | Map) | (Disc | Sunbeam)) | ((Marion | Nepal) | Tanis))

[5] (Map → Well | Disc → Sunbeam) | (Nepal → Marion | Tanis)

[6] Disc → Map → Sunbeam → Well | Nepal → Marion → Tanis

[7] Nepal → Marion → Disc → Tanis → Map → Sunbeam → Well

Figure 6.5 Mergesort splits a list into two sublists of equal size, sorts them, and merges the sorted results into one sorted list. The parentheses indicate the order in which lists must be merged. In line 4 the decomposition is complete when only single-element lists are obtained. In line 5 three pairs of single-element lists have been merged into three sorted two-element lists, and in line 6 those lists are merged again into one four-element and one three-element list, which are merged in the last step to produce the final result.

0	2	0	2	1	1	0	···	*counter*
1	2	3	4	5	6	7	···	*index*

Diagram 25

Table 7.1 Approximate runtimes for different sizes of input on a computer that can perform a billion steps per second.

Input Size	Runtime			
	Linear	**Linearithmic**	**Quadratic**	**Exponential**
20				0.001 second
50				13 days
100				
1,000				
10,000			0.1 second	
1 million	0.001 second	0.002 second	16 minutes	
1 billion	1 second	30 seconds	32 years	

Note: Blank cells indicate runtimes < 1 millisecond, too small to be meaningful for human perception of time, and thus of no practical significance. Gray cells indicate runtimes too immense to comprehend.

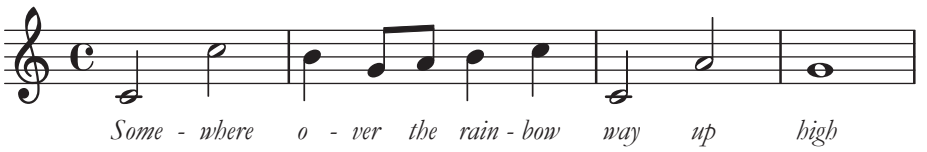


Diagram 26

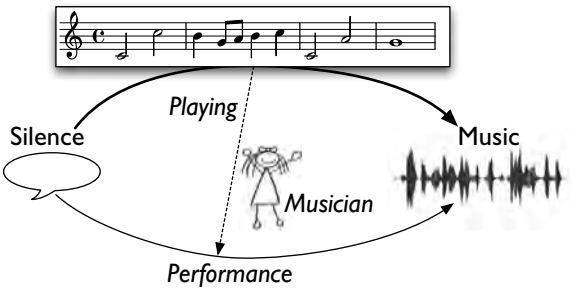


Figure 8.1 Playing (executing) a music s core (an algorithm) generates a performance (a computation), which transforms s ilence into music. The performance is produced by a musician (a computer), f or example, a person or a machine that can understand the music notation (language) in which the piece is written. See figure 2.1 on page 15

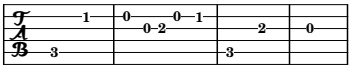


Diagram 27

Table 8.1

Grammars	Equations	Algorithms
Nonterminal	Variable	Parameter
Terminal	Constant, operation	Value, instruction
Sentence	Fact	Algorithm with only values
Rule		Instruction

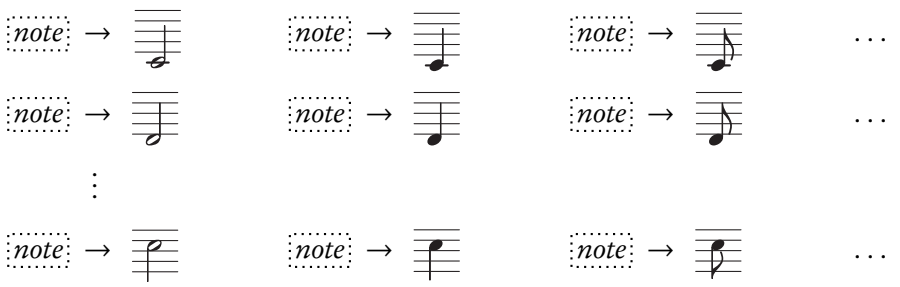


Figure 8.2 Grammar rules defining the possible substitutions for the note nonterminal. Since an arbitrary note is represented by one nonterminal, a separate rule is required for each combination of pitch and duration. Each column shows the rules for notes for a particular duration: *left*, half notes (notes that last one half of a measure); *middle*, quarter notes; *right*, eighth notes.

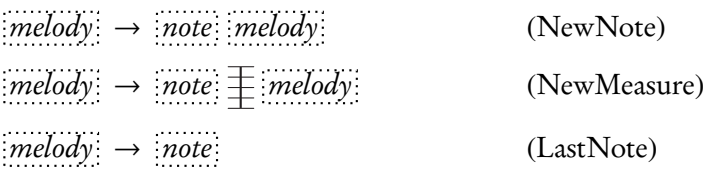

melody → note melody (NewNote)
melody → note melody (NewMeasure)
melody → note (LastNote)

Diagram 28

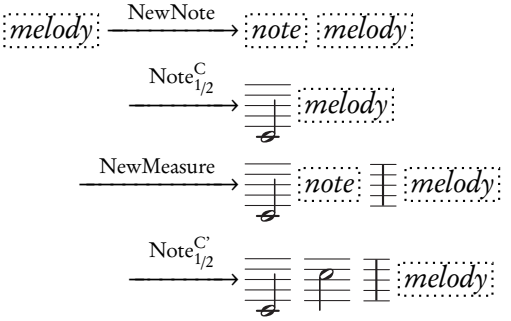

melody
NewNote
note
melody
Note C 1/2
melody
NewMeasure
note
melody
Note C' 1/2
melody

Diagram 29

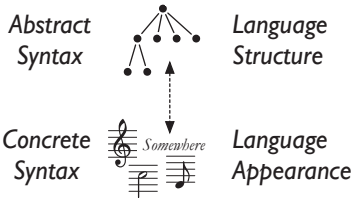

Abstract
Syntax
Language
Structure
Concrete
Syntax
Somewhere
Language
Appearance

Diagram 30

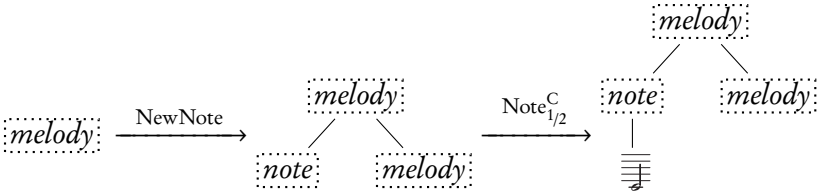

melody
NewNote
melody
note
melody
Note$^{C}_{1/2}$
melody
note
melody

Diagram 31

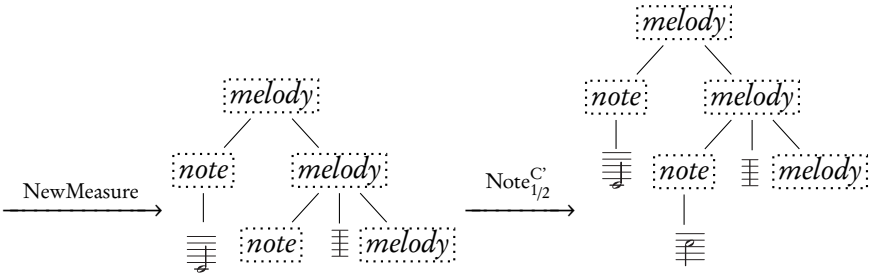

melody
note
melody
note
melody
NewMeasure
Note$^{C'}_{1/2}$
melody
note
melody
note
melody

Diagram 32

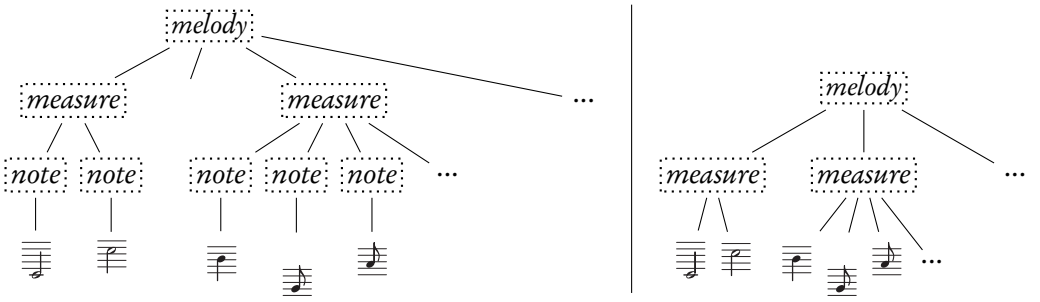


Figure 8.3 The structure of a melody represented by syntax trees. A syntax tree captures the structural result of a derivation but ignores details, such as which rules have been applied in which order. *Left:* A parse tree, which captures all the details of the derivation. *Right:* An abstract syntax tree, which omits unnecessary details and retains only the structurally relevant information.

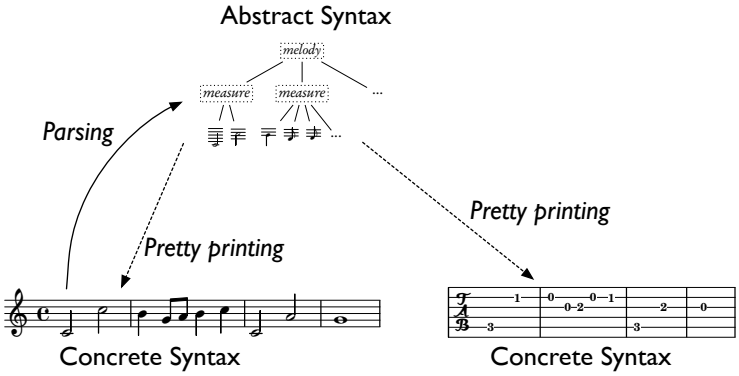


Figure 8.4 Parsing is the process of identifying the structure of a sentence and representing it as a syntax tree. Pretty printing turns a syntax tree into a sentence. Since an abstract syntax tree may omit certain terminal symbols (for example, bars), pretty printing cannot just collect the terminals in the leaves of the syntax tree but must generally employ grammar rules to insert additional terminal symbols. The parsing arrow is missing for the tablature notation because it is an ambiguous notation and thus doesn't allow the construction of a unique abstract syntax tree.

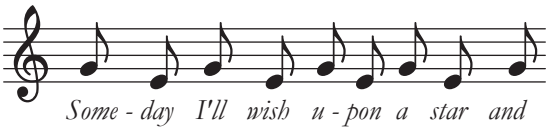


Diagram 32b

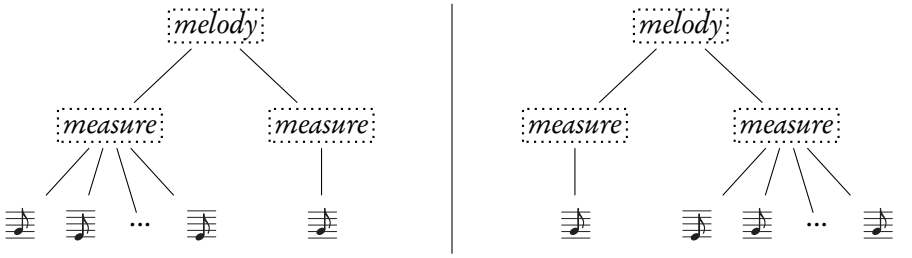


Figure 9.1 Ambiguity in a grammar can cause a sentence to have different abstract syntax trees. These trees present a different hierarchical structure for the same sentence. It is generally impossible to determine the meaning of such a sentence, since its structure affects its meaning.

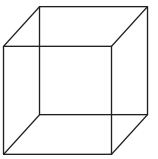


Diagram 33

Diagram 34

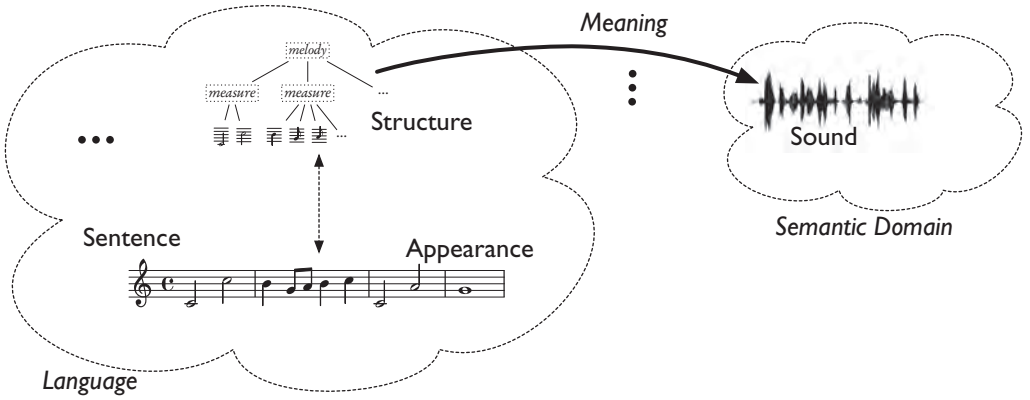


Figure 9.2 The semantics of a language is given by a mapping that associates each sentence's structure, represented by its abstract syntax tree, with an element of the semantic domain. This view of meaning is called *denotational semantics*, since it is based on assigning denotations to sentences of a language.

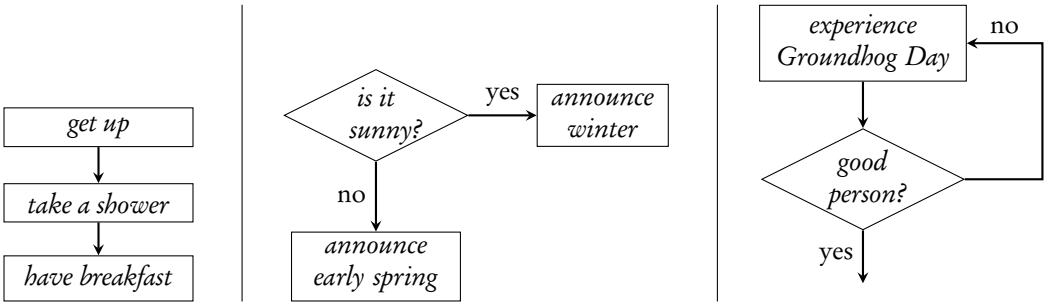


Figure 10.1 Flowchart notation for control structures. *Left:* A sequence of steps Phil Connors takes every morning. *Middle:* A conditional showing the decision that Punxsutawney Phil faces on each Groundhog Day. *Right:* A loop expressing the life of Phil Connors during the movie *Groundhog Day*. The "no" arrow and the arrow to the condition form a cycle through the two nodes.

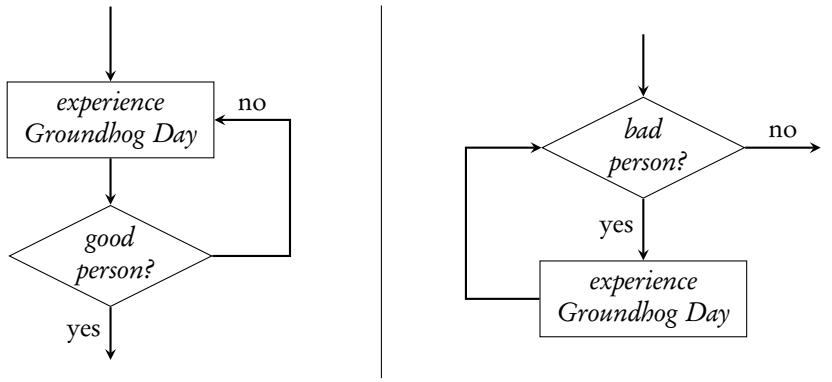


Figure 10.2 Flowchart notation illustrating the different behavior of repeat and while loops. *Left:* Flowchart of a repeat loop. *Right:* Flowchart of a while loop.

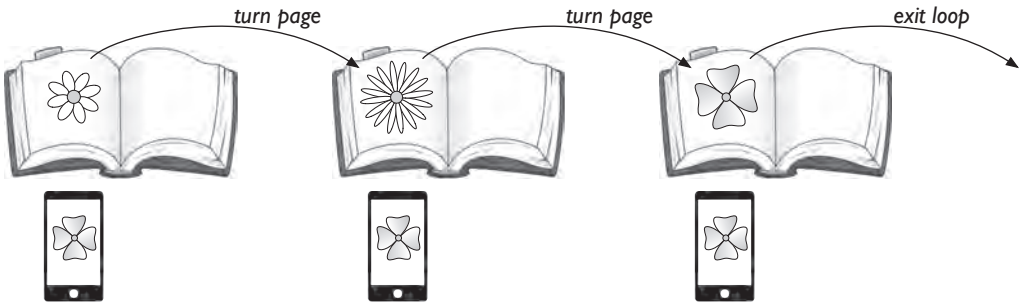


Figure 11.1 Unfolding a loop means to create a sequence of copies of its body. For each iteration of the loop a separate copy of the loop body is created. The figure shows the state that is modified and how it changes (if at all). If the state never changes so that the termination condition is fulfilled, the sequence becomes infinite.

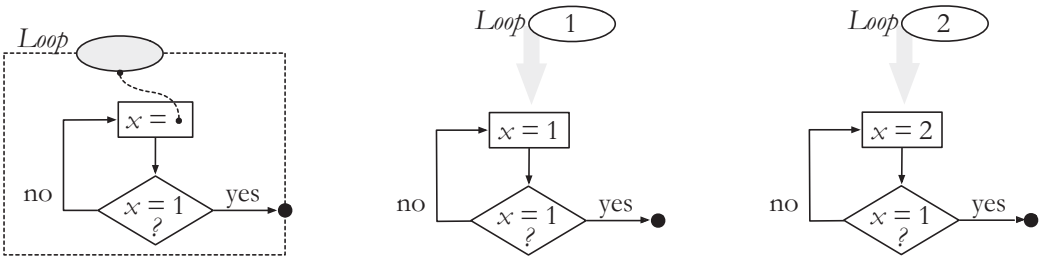


Figure 11.2 The algorithm *Loop* stops when called with the number 1 as an argument but loops forever when called with any other argument. *Left:* The definition of the algorithm *Loop*. *Middle:* Application of *Loop* to the number 1. *Right:* Application of *Loop* to the number 2.

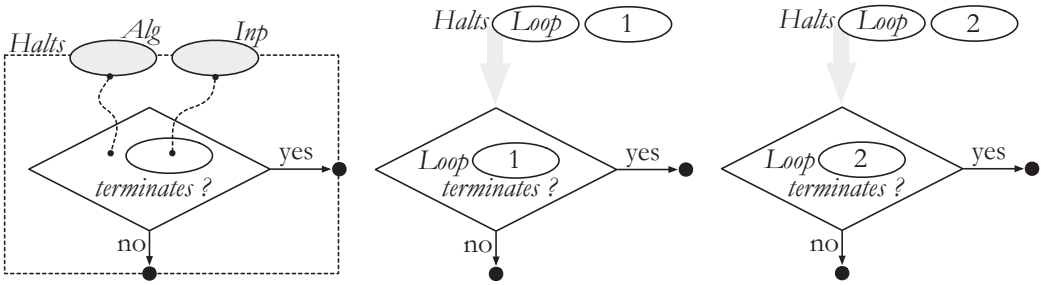


Figure 11.3 *Left:* The structure of the algorithm *Halts*. It takes two parameters, an algorithm (*Alg*) and its input (*Inp*), and tests whether the algorithm applied to the input *Alg*(*Inp*) terminates. *Middle:* Application of *Halts* to the algorithm *Loop* and number 1, which results in "yes." *Right:* Application of *Halts* to *Loop* and the number 2, which results in "no."

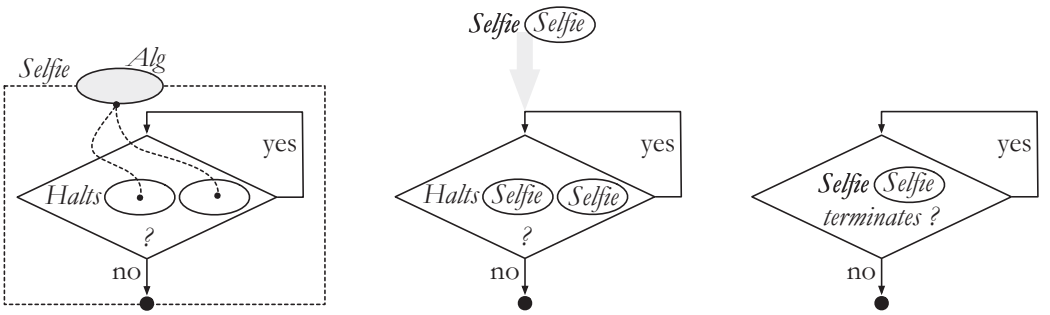


Figure 11.4 *Left:* The definition of the algorithm *Selfie*. It takes an algorithm (*Alg*) as a parameter and tests whether the algorithm applied to itself terminates. In that case *Selfie* will enter a nonterminating loop. Otherwise, it will stop. *Middle:* Application of *Selfie* to itself leads to a contradiction: If *Selfie* applied to itself terminates, then it goes into a nonterminating loop, that is, it doesn't terminate; if it doesn't terminate, it stops, that is, it terminates. *Right:* Expanding the definition of *Halts* provides a slightly different view of the paradox.

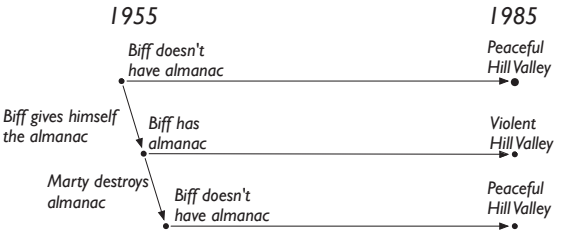

1955
1985
Biff doesn't have almanac
Peaceful Hill Valley
Biff gives himself the almanac
Biff has almanac
Violent Hill Valley
Marty destroys almanac
Biff doesn't have almanac
Peaceful Hill Valley

Diagram 35

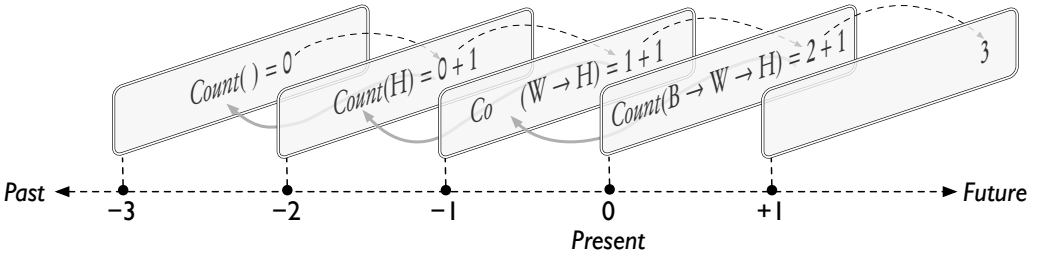


Figure 12.1 Recursively counting the number of elements in a list by traveling into the past. The execution of the algorithm *Count* on a nonempty list triggers a computation that adds 1 to the result of the computation of *Count* for the list's tail. Executing the computation of the tail one step in the past makes its result available in the present, and the result of the addition will be available one step in the future. Executing *Count* on a nonempty list in the past leads to a further execution of *Count* even further back in the past.

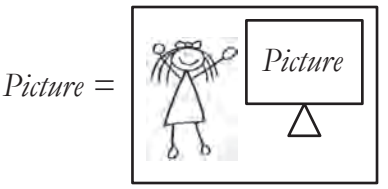


Figure 13.1 Recursive picture definition. A name is given to a picture, which contains the name and thus a reference to itself. The meaning of such a self-referential definition can be obtained by repeatedly substituting a scaled-down copy of the picture for its name, thus producing step-by-step an unfolded recursion.

$$
\begin{aligned}
\mathit{Count}(\mathrm{B}{\rightarrow}\mathrm{W}{\rightarrow}\mathrm{H}) &\xrightarrow{\mathit{Count}_2} \mathit{Count}(\mathrm{W}{\rightarrow}\mathrm{H}) + 1 \\
&\xrightarrow{\mathit{Count}_2} \mathit{Count}(\mathrm{H}) + 1 + 1 \\
&\xrightarrow{\mathit{Count}_2} \mathit{Count}(\,) + 1 + 1 + 1 \\
&\xrightarrow{\mathit{Count}_1} 0 + 1 + 1 + 1
\end{aligned}
$$

Diagram 36

Goal(*live now*)

$\xrightarrow{Goal_1}$ *Goal*(*restore world*); *Go camping*

$\xrightarrow{Goal_2}$ *retrieve sports almanac*; *Goal*(*save Doc*); *go camping*

$\xrightarrow{Goal_3}$ *retrieve sports almanac*; *help Doc avoid Buford Tannen*; *go camping*

Diagram 37

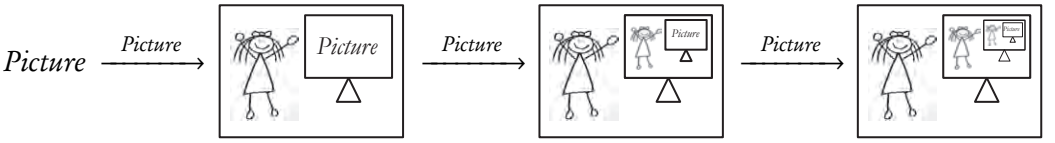


Diagram 38

$$Ones \xrightarrow{Ones} 1{\rightarrow}Ones \xrightarrow{Ones} 1{\rightarrow}1{\rightarrow}Ones \xrightarrow{Ones} 1{\rightarrow}1{\rightarrow}1{\rightarrow}Ones \cdots$$

Diagram 39

$$
\begin{aligned}
&\mathit{Isort}(\ ,\mathit{list}) &&= \mathit{list}\\
&\mathit{Isort}(x\rightarrow \mathit{rest},\mathit{list}) &&= \mathit{Isort}(\mathit{rest},\mathit{Insert}(x,\mathit{list}))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Insert}(w,\ ) &&= w\\
&\mathit{Insert}(w,x\rightarrow \mathit{rest}) &&= \textbf{if } w \le x \textbf{ then } w\rightarrow x\rightarrow \mathit{rest} \textbf{ else } x\rightarrow \mathit{Insert}(w,\mathit{rest})
\end{aligned}
$$

Diagram 40

$$
\begin{aligned}
\mathit{Isort}(\mathrm{B}{\to}\mathrm{W}{\to}\mathrm{H},\ ) &\xrightarrow{\mathit{Isort}_2} \mathit{Isort}(\mathrm{W}{\to}\mathrm{H}, \mathit{Insert}(\mathrm{B},\ )) \\
&\xrightarrow{\mathit{Insert}_1} \mathit{Isort}(\mathrm{W}{\to}\mathrm{H}, \mathrm{B}) \\
&\xrightarrow{\mathit{Isort}_2} \mathit{Isort}(\mathrm{H}, \mathit{Insert}(\mathrm{W}, \mathrm{B})) \\
&\xrightarrow{\mathit{Insert}_2} \mathit{Isort}(\mathrm{H}, \textbf{if } \mathrm{W} \leq \mathrm{B} \textbf{ then } \mathrm{W}{\to}\mathrm{B} \textbf{ else } \mathrm{B}{\to}\mathit{Insert}(\mathrm{W},\ )) \\
&\xrightarrow{\textbf{else}} \mathit{Isort}(\mathrm{H}, \mathrm{B}{\to}\mathit{Insert}(\mathrm{W},\ )) \\
&\xrightarrow{\mathit{Insert}_1} \mathit{Isort}(\mathrm{H}, \mathrm{B}{\to}\mathrm{W}) \\
&\xrightarrow{\mathit{Isort}_2} \mathit{Isort}(\ , \mathit{Insert}(\mathrm{H}, \mathrm{B}{\to}\mathrm{W})) \\
&\xrightarrow{\mathit{Isort}_1} \mathit{Insert}(\mathrm{H}, \mathrm{B}{\to}\mathrm{W}) \\
&\xrightarrow{\mathit{Insert}_2} \textbf{if } \mathrm{H} \leq \mathrm{B} \textbf{ then } \mathrm{H}{\to}\mathrm{B}{\to}\mathrm{W} \textbf{ else } \mathrm{B}{\to}\mathit{Insert}(\mathrm{H}, \mathrm{W}) \\
&\xrightarrow{\textbf{else}} \mathrm{B}{\to}\mathit{Insert}(\mathrm{H},\mathrm{W}) \\
&\xrightarrow{\mathit{Insert}_2} \mathrm{B}{\to}(\textbf{if } \mathrm{H} \leq \mathrm{W} \textbf{ then } \mathrm{H}{\to}\mathrm{W} \textbf{ else } \mathrm{W}{\to}\mathit{Insert}(\mathrm{H}, \underline{\mathit{rest}})) \\
&\xrightarrow{\textbf{then}} \mathrm{B}{\to}\mathrm{H}{\to}\mathrm{W}
\end{aligned}
$$

Figure 13.2 Substitution trace for the execution of insertion sort.

Unsorted List		Sorted List
B→W→H		
W→H		B
H		B→W
		B→H→W

Diagram 41

Current Instruction	Stack	State of the World
① *ToDo*(1955)	-	*Year*: 1985, *Biff has almanac*, *Mayhem in* 1985
④ *destroy almanac*	②	*Year*: 1955, *Biff has almanac*, *Mayhem in* 1985
⑤ *ToDo*(1885)	②	*Year*: 1955, *Doc in danger in* 1885
⑦ *save Doc*	⑥ ②	*Year*: 1885, *Doc in danger in* 1885
⑧ *return*	⑥ ②	*Year*: 1885
⑥ *return*	②	*Year*: 1955
② *go camping*	-	*Year*: 1985

Figure 13.3 Interpretation of *ToDo*(1985). If the current instruction is a recursive call, the address following it is remembered on the stack to allow the continuation of the computation after it is completed. This happens whenever a *return* instruction is encountered. After the jump back, the address is removed from the stack.

ToDo(1985) = ① *ToDo*(1955) ② *go camping* ③ *return*
ToDo(1955) = ④ *destroy almanac* ⑤ *ToDo*(1885) ⑥ *return*
ToDo(1885) = ⑦ *save Doc* ⑧ *return*

Diagram 42

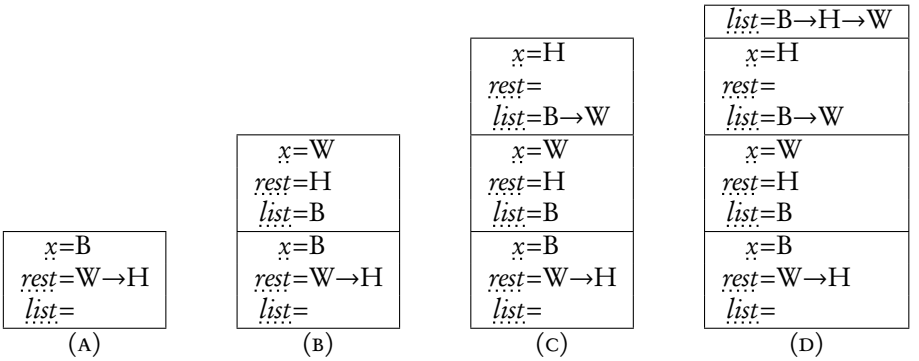


Figure 13.4 *Snapshots of stack values during the interpretation of Isort*(B→W→H,).

$$Qsort(\,) \quad =$$
$$Qsort(x \rightarrow rest) = Qsort(Smaller(rest, x)) \rightarrow x \rightarrow Qsort(Larger(rest, x))$$

Diagram 42b

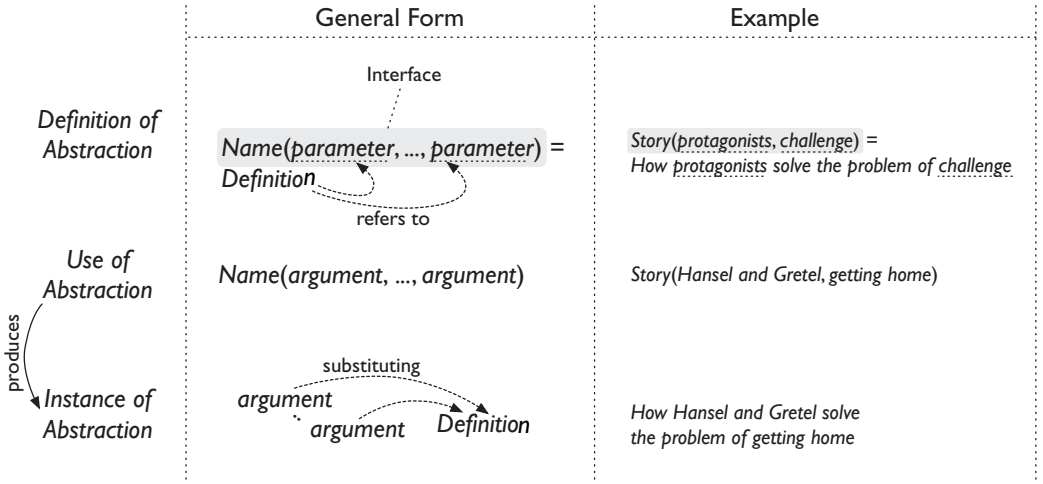


Figure 15.1 Definition and use of abstractions. The definition of an abstraction assigns it a name and identifies parameters that are referred to by its definition. The name and parameters are the abstraction's interface, which prescribes how the abstraction is to be used: through its name and by supplying arguments for its parameters. Such a use generates an instance of the abstraction, which is obtained by substituting the arguments for the parameters in the definition of the abstraction.

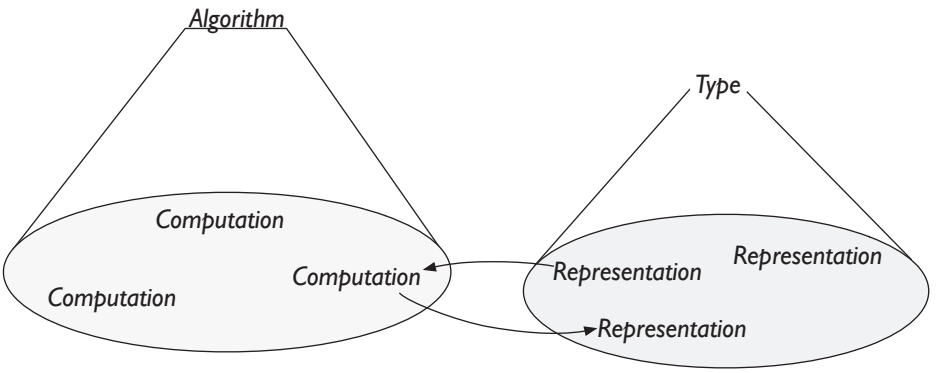


Figure 15.2 An algorithm is an abstraction from individual computations. Each algorithm transforms representations. A type abstracts from individual representations. If *Input* is the type of representations accepted by an algorithm, and *Output* is the type of representations produced by it, then the algorithm has the type *Input* → *Output*.

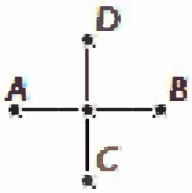


Diagram 43

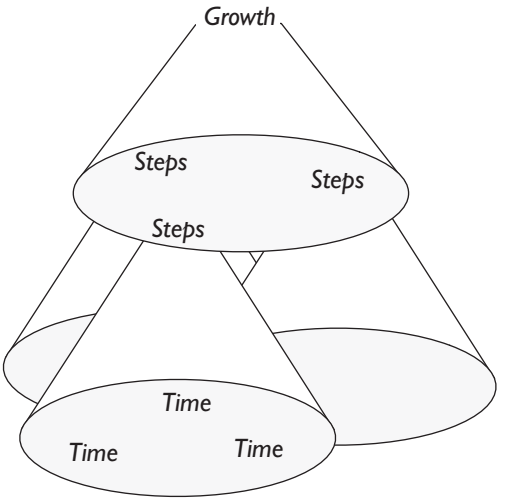


Figure 15.3 Time abstraction. To abstract from the different speeds of computers, we use the number of steps an algorithm needs during execution as a measure for its runtime. To abstract from the different number of steps needed for different inputs, we measure the runtime of an algorithm in terms of how fast it grows for larger input.

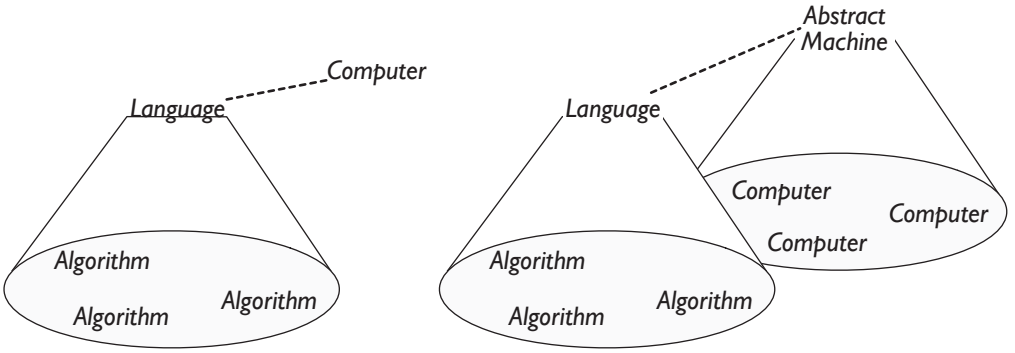


Figure 15.4 An abstract machine is an abstraction from concrete computers. By providing a simpler and more general interface, an abstract machine makes algorithmic languages independent of specific computer architectures and extends the range of computers that can execute them.

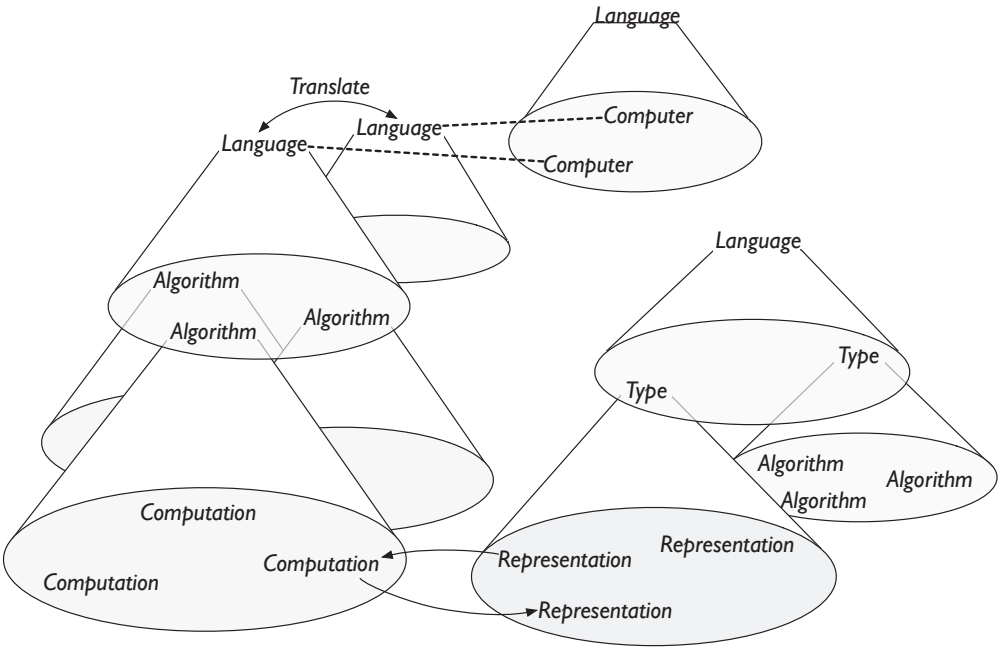


Figure 15.5 The tower of abstractions. An algorithm is a (functional) abstraction from computations. An algorithm transforms representations, whose (data) abstractions are types. The acceptable inputs and outputs for an algorithm are also expressed as types. Each algorithm is expressed in a language, which is an abstraction from algorithms. A translation algorithm can transform algorithms from one language into another and therefore makes algorithms independent from a particular computer or abstract machine that understands the language in which it is expressed. Language is also an abstraction of computers, since translation can effectively eliminate the differences between computers. Types are also expressed as part of a language. The abstraction hierarchy illustrates that all abstractions in computer science are expressed in some language.