# Problem Solving via Abstraction

Tom Akita
tomakita@gmail.com

October 22, 2017

**Abstract**

In this paper, a set of problem solving tools is introduced with the objective of aiding in the solution of software engineering problems. In the process, definitions of common software engineering terms such as ambiguity, elegance, and generality are developed.

## Introduction

It is common for a software engineer to spend a great deal of time solving variations of the same problem. For example, a full-stack web engineer may regularly draft solutions of the following pattern: query a database, serialize the data that is returned for transmission over a network, deserialize the data, and then display the data on a webpage. Repetitive work of this sort can encourage problem solving by pattern matching: we use roughly the same pattern of steps to solve each problem, because most of the problems we need to solve are roughly the same. Rather than problem solving, this is more accurately thought of as a process of framing problems in terms of some predetermined solution pattern. The degree to which this approach is successful can be measured by the similarity between the problem being solved and the problem for which the pattern was developed. Thus, while intuitive, this approach to problem solving is not flexible, and will fail when applied to classes of problems for which the pattern is not suited.

In this paper, we attempt to retain the intuitive appeal of this approach to problem solving while improving its flexibility by considering attributes which are common to many problems, and then developing a set of tools

which can be applied to any problem possessing these attributes. This set of tools is based upon the notion of abstraction.

# Motivation

The principle difficulty in the development of a general problem solving pattern is the heterogeneity of the problems such a pattern must be applicable to. Our first task in the development of such a pattern, then, should be to come up with a way to view all problems in the same terms. One such way is via abstraction.

Abstraction is likely to be a familiar, if imprecisely-defined, concept to software engineers. Words beginning with "abstract-" have a variety of different usages:

| | |
|---|---|
| Adverb: | *My henchmen are behaving abstractly today. It must be due to the full moon.* |
| Noun: | *The abstraction of the pipe organ is one of power and pain.* |
| Verb: | *All of my brains are the same if you abstract away all of their differences.* |

This paper is concerned with the verb form of abstraction. Here are two intuitive definitions of this form of abstraction:

> *The essence of abstraction is to extract essential properties while omitting inessential details.*[RGI75]

> *Abstraction is the selective examination of certain aspects of a problem. The goal of abstraction is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant.*[RBP+91]

Abstraction is usually regarded as the act of removing unnecessary detail from consideration. If we generalize this definition to say that abstraction is the act of adding necessary or removing unnecessary detail from consideration, we can see how this might be a useful tool for problem solving, because

the difficulty of problem solving is largely a matter of scope. A particular problem may have an infinite number of aspects: countless details, countless different ways of interpreting those details, and countless different ways of using those details to derive a solution. Abstraction allows us to narrow or widen this scope as needed. It allows us to reason in terms of a group of hench*men* instead of in terms of each individual hench*man* (and vice-versa), and to construct solutions which are both elegant and correct.

# Definitions

In order to understand more clearly the ways in which abstraction may be applied to any problem, we create a small, informal language, and use it to define abstraction. In particular, we frame abstraction as a pair of operations, $+$ and $-$, which add and remove information from some domain of discourse. We call this domain of discourse a context. Information in a context is represented by recursively-defined **expressions**. Just as information can be thought of as a composition of smaller pieces of information, expressions can be thought of as a composition of smaller expressions.

| | | |
|---|---|---|
| $\langle expression \rangle$ | ::= | $\langle expression \rangle \circ \langle expression \rangle$ |
| | \| | `term` |
| `term` | ::= | anything |

Figure 1: The syntax of an expression.

We give the syntax of an expression in *Backus-Naur Form* (a/k/a BNF, see [ALSU06] for more information), in which the ::= symbol means "is defined as" and the | symbol means "or"; i.e. an expression is defined either as a composition (via some operation $\circ$) of expressions or as a *term*. An identifier surrounded by angle brackets ($\langle$, $\rangle$) means that it is recursively-defined (in BNF, this is called a *nonterminal symbol*), whereas the absence of angle brackets means that it is not recursively-defined (in BNF, this is called a *terminal symbol*).

As for semantics, we place no restrictions on how expressions may be composed, nor do we place any restrictions on what a term may be. That is, $\circ$ may stand for any binary operation, and a term may be anything that is not recursively defined. For this reason, the language being developed is not a formal language – instead, it may be likened to a pseudocode language.

| | | |
|---|---|---|
| morris is a cat | $\Longleftarrow$ | morris is a furry thing $\underline{\text{and}}$ all furry things are cats |
| 5 | $=$ | $3 \underline{+} 2$ |

Figure 2: Logical implication and addition: two examples of expression relations. In each relation, the composition operator is underlined, and the relational operator has a darkened background. What is on the left of each relational operator is an expression, and what is on the right of each relational operator is composed to make an expression.

When writing $d = e \circ f$, we say that we *relate* expression $d$ with the composite expression $e \circ f$ via some relation $=$. A problem will generally involve a large number of expressions whose relations with one another can be conveniently represented by an indentation-delimited **scope**. The scope of an expression contains all of the expression's subexpressions, recursively.

Because abstraction involves the addition/removal of necessary/unnecessary expressions from such a scope, there must be some way of inferring which expressions are necessary/unnecessary. For this purpose, we specify a **goal**, given in natural language, which defines what the expressions in the scope should be used for. Together, an expression scope and a goal make a **context**. Note that a context's expression scope may initially contain a single expression of vacuous meaning – e.g. `null`, effectively serving as the root scope of the context. Given a context $c$, the purpose of abstraction is to use the expressions in $c$'s scope to reason toward the satisfaction of the $c$'s goal.

```
goal = do something with the expressions in the scope

expression
   expression
   expression
      expression
expression
```

Figure 3: A goal and scope representing a context. In this paper, contexts are written over a darkened background.

Abstraction occurs via two unary operations, $+$ and $-$, which can be applied to a context: $+$ adds one or more expressions of our choosing to a context, and $-$ removes one or more expressions of our choosing from a context. Intuitively, the goal of a context informs our use of $+$ and $-$ in

adding and removing expressions from the context. + is applied to a context when the context contains so few expressions that we don't know how to reason toward the satisfaction of the goal. − is applied when the context contains so many expressions that it is simply too difficult to know which will be useful in reasoning toward the satisfaction of the goal.

In software engineering, some languages are meant to represent transformations to be executed as programs, and some languages are meant to represent data to be parsed as documents. The language introduced in this paper is meant to represent hierarchically-defined data to be reasoned with as a context. This reasoning is facilitated by + and −, i.e. by abstraction.

# Example: Linked List Computation

In this section, we consider two different contexts and their images under +:

```
goal = compute the length of a linked list

(empty scope)
```

+↓

```
goal = compute the length of a linked list

linked list
```

+↓

```
goal = compute the length of a linked list

linked list
   each node has a successor
```

Figure 4: An application of + to a context. If we would like to compute the length of a linked list, it would be useful to know that each node has a successor, as this is what enables traversal of the list.

If the **size** of a context is measured by the number of expressions in its scope, then + increases the size of the context, and − decreases the size of the context. In this way, abstraction allows us to change the size of a context, which allows us to control the number of expressions with which we can reason. Given two contexts with identical goals but different sizes, we say

that the smaller context allows us to reason at a *higher level of abstraction*, and the larger context allows us to reason at a *lower level of abstraction*.

```
goal = compute the sum of the members of a linked list

(empty scope)
```

$+\downarrow$

```
goal = compute the sum of the members of a linked list

linked list
```

$+\downarrow$

```
goal = compute the sum of the members of a linked list

linked list
   each node has a successor
   each node has a value
```

Figure 5: Another application of +.

When applying + or − to a context, expressions are added to or removed from any existing expression. However, the addition or removal of some expressions will inevitably be more helpful than the addition or removal of others in reasoning toward the satisfaction of a goal. For example, adding the `each node has a value` expression to the `compute the length of a linked list` context probably wouldn't be useful – at least not in an obvious way, as doing so probably wouldn't help us to satisfy our goal of computing the length of a linked list. This raises the issue of which applications of + and − are worthwhile, and which are not.

We say that an expression is **valid** if its presence in the context may conceivably help us to satisfy the goal. This is a vague notion, but it will be given more meaning in the next section. If all of a context's expressions are valid, we say that the context itself is valid. And if two contexts with identical goals are both valid, we say that they are **equivalent**.

Thus, the `each node has a value` expression is not valid in the `compute the length of a linked list` context, and if we were to add it to said context, the resulting context would be in a different **equivalence class** from the original context. In the next section, in which problem solving is framed in terms of abstraction, we will see that if two contexts are in different

equivalence classes, it means that they correspond to different problems. It won't always be clear whether or not an application of + or − will result in a context that is a member of a different equivalence class, and the degree to which this is the case is called the context's **ambiguity** with respect to the expression being added or removed. A context's ambiguity with respect to *all* of its expressions is defined as the aggregate of its ambiguity with respect to each of its expressions. For example, in the contexts above, we assumed that each node in a linked list has a pointer to the next node. Since these contexts are not accompanied by any explicit definition of a linked list, we've had to make this assumption, and it is somewhat ambiguous whether or not this assumption is valid, i.e. whether or not the resulting contexts are valid. Thus, an **assumption** is an application of + or − that may possibly result in an invalid context.

# Problem Solving

We frame problems as objects to which abstraction can be applied. Problems of the sort that software engineers solve are generally specified as (or can be converted to) a natural language problem statement which will involve *things*, *statements about things*, and *what we can do with things*. A solution is expected to be given as some procedure, along with justification which explains why that procedure is a solution to the problem. In this paper, we do not consider the implementation of such solutions (see [Wir71], [Hoa69], [Dij75], [WF67], [Gri87], and [SM09] for pointers on this subject), we merely consider their derivation.

We refer to *things* and *statements about things* as **facts**, and we refer to *what we can do with things* as **actions**. We begin by giving more precise definitions for the notions of fact and action, along with a few related definitions.

Representing *things* and *statements about things*, facts are truth statements whose expression operators are `and` and `or`.

| | | |
|---|---|---|
| ⟨*conjunction*⟩ | ::= | ⟨*fact*⟩ `and` ⟨*fact*⟩ |
| ⟨*disjunction*⟩ | ::= | ⟨*fact*⟩ `or` ⟨*fact*⟩ |
| ⟨*fact*⟩ | ::= | ⟨*conjunction*⟩ |
| | \| | ⟨*disjunction*⟩ |
| | \| | `true` |
| | \| | `false` |
| **ex fact**: | | |
| is giraffe | = | is 5'10" and is 150 lbs |

Figure 6: Syntax of the fact expression type.

Representing *what we can do with things*, actions are a variation of the *Hoare triple* (introduced in [Hoa69]) whose pre- and post- conditions are facts, and whose expression operator is ;. In a nutshell, a Hoare triple is meant to convert a procedure into a truth statement by qualifying a procedure with pre- and post- conditions. If the precondition is true before the procedure executes, then the postcondition will be true after the procedure executes. A **signature** is an action whose **procedure** is yet to be defined.

| | | |
|---|---|---|
| ⟨*action*⟩ | ::= | {precondition} ⟨*action*⟩ ; ⟨*action*⟩ {postcondition} |
| | \| | {precondition} `procedure` {postcondition} |
| | \| | `signature` |
| `signature` | ::= | {precondition} `undefined` {postcondition} |
| `procedure` | ::= | some transformation that, given a precondition, implies a postcondition |
| **ex action**: | | |
| eat | = | **{the input is tasty}**<br>chew ; swallow<br>**{the input is no longer tasty}** |

Figure 7: Syntax of the action expression type.

Representing the validity of an action in some context, a **solution** relates a justification (given as a composition of facts) to an action via logical implication.

| | | |
|---|---|---|
| $\langle solution \rangle$ | ::= | $\langle solution \rangle$ ; $\langle solution \rangle$ |
| | \| | $\langle fact \rangle \implies \langle solution \rangle$ |
| | \| | $\langle fact \rangle \implies \langle action \rangle$ |
| **ex solution**: | | |
| is a cat | = | anything that is furry is a cat<br>$\implies$<br>**{the input is something that may be furry}**<br>if the input is furry, then it is a cat. otherwise, it is not a cat.<br>**{the input is or is not a cat}** |

Figure 8: Syntax of the solution expression type. $\implies$ is the logical implication operator.

Facts, actions, and solutions are all types of expressions. When problem solving via abstraction, the validity of a context is determined by the values that its expressions resolve to – i.e. to either `true` or `false`. By making facts, actions, and solutions all resolve to this same pair of values, checking a context for validity is made simple.

signature = **{input}** `undefined` **{output}**

context =

```
goal = a solution for signature

some expression representing information in the problem
    statement
some expression representing information in the problem
    statement
...
```

Figure 9: Together, a signature and a context represent a problem. In the remainder of this paper, the goal will always be the same (deriving a solution for a given signature), so we elect not to write the goal in the context.

A natural language problem statement is converted to an assortment of facts along with a signature. The objective of the problem is to define the procedure of this signature, as well as to give a composition of facts which justifies the procedure's correctness. In other words, we wish to use the given facts to derive a solution involving the given signature. A **problem**, then, is an encoding of a natural language problem statement as a context whose

expression scope is a fact scope, and whose goal is to derive a solution for some signature that is given in the problem statement.

## Problem: Linked List Search

| | |
|---|---|
| problem statement: | A linked list is a collection of *nodes*. This collection can be accessed via its first node, called the *head*. Each node in the collection has a pointer to the next node in the collection. Given a linked list $l$ of integers, find the node in $l$ with greatest value. |

From this problem statement, we create a signature and a context so that we may use abstraction to derive a solution to the resulting problem.

| | |
|---|---|
| signature = | **{ $l$ is a linked list }**<br>`undefined`<br>**{ the node with greatest value in $l$ is returned }** |

context =

```
linked list
```

Figure 10: The problem of searching through a linked list.

The derivation of a solution to this problem can happen at different levels of abstraction, depending upon the engineer's level of familiarity with the facts contained in the problem's context. For example, an engineer who is familiar with linked lists and integers may immediately come up with the following sketch of a solution:

> *Traverse the nodes of the linked list, keeping track of the node n with greatest value as the traversal occurs. When the traversal is over, return n.*

However, an engineer who is unfamiliar with linked lists is unlikely to know offhand how to traverse them (or that they can even be traversed), so this solution is defined at a level of abstraction that is too high to be of use to such an engineer. In general, we seek to derive solutions at a

level of abstraction that is just low enough to be tractable – i.e. at a level of abstraction that is low enough to enable us to reason our way toward a solution, and no lower. This enables us to reason using just as much information as we require, and no more. If we don't already know how to traverse a linked list, we can apply + to the problem's context, in hopes that the added detail will give us clues as to how such a traversal might be done.

```
linked list
```

$+\downarrow$

```
linked list
   has a head
   each node has a successor
```

Figure 11: Applying + to the problem context.

Since the list has a head, and each node has a successor, the entire list should be reachable from the head of the list. This suggests a traversal strategy which involves beginning from the head, and visiting each node's successor until there are no more nodes to visit. If we could somehow use this traversal strategy to find the node with greatest value, we would have a solution to our problem. Thus, we supply this as a solution, and give the part that we are unsure about (finding the greatest node during the traversal) as a signature.

```
The list l has a head and each node has a successor, which means
   that the entire list is reachable from the head of l.
⟹
{l is a linked list}
   Traverse l by beginning at the head and accessing each node's
      successor until the end of the list is reached. During
      the traversal...
   ;
   {l is a linked list, g is the current node in the traversal}
      check to see if g is the greatest in l
   {the node with greatest value in l is returned}
{the node with greatest value in l is returned}
```

Figure 12: A solution, with one of the actions given as a signature (identified by the `tyson futuristic cyborg` typeface).

Since part of the solution is given as a signature (and is thus partially undefined), we must find a definition for this signature's procedure before we can consider the original problem to be solved. We do this via a **subproblem**, which is simply a problem whose solution is part of another solution. A subproblem inherits a subset of the facts in the parent problem's scope, and has as goal some signature that is used in the parent problem's solution.

Solving problems by division into subproblems yields a context in which the scope is a scope of solutions, and the goal is to derive a complete (i.e. without the use of signatures) definition for the solution at the root level of the scope. The definition of each solution in the context is dependent upon the definitions of each subsolution in its scope, and we work to derive solutions from left to right in this scope. In other words, division of a problem into subproblems allows us to solve problems in a *top-down* manner – it allows us to first solve the problem at a high level of abstraction, and then at increasingly lower levels of abstraction, which allows us to limit the size of the context associated with each subproblem. Thus, the use of subproblems allow us to treat fact contexts as if they were expressions, albeit indirectly: a solution is a composition of less abstract solutions, and each of those solutions is derived using its own fact context.

Here is the subproblem at hand:

| | |
|---|---|
| | **{ $l$ is a linked list, $g$ is the current node in the traversal }** |
| signature = | `undefined` |
| | **{ the node with greatest value in $l$ is returned }** |

context =

```
linked list
   has n nodes
      each node has an integer value
```

Figure 14: The subproblem of finding the greatest value during a linked list traversal.

This subproblem asks that we come up with a way of determining whether or not a particular node has the greatest value in the list. If we have trouble coming up with ideas, we can apply + to the problem scope again in hopes that the resulting increase in detail will be revelatory (see Figure 15).

```
root problem (a/k/a problem 0) signature
```

solve root problem (via +) ↓

```
justification for root problem
⟹
{precondition 0}
   subproblem 1a signature ; subproblem 1b signature
{postcondition 0}
```

solve problem 1a (via +) ↓

```
justification for root problem
⟹
{precondition 0}
   justification for subproblem 1a
   ⟹
   {precondition 1a}
      procedure 1a definition
   {postcondition 1a}
   ;
   subproblem 1b
{postcondition 0}
```

solve problem 1b (via +) ↓

```
justification for root problem
⟹
{precondition 0}
   justification for subproblem 1a
   ⟹
   {precondition 1a}
      procedure 1a definition
   {postcondition 1a}
   ;
   justification for subproblem 1b
   ⟹
   {precondition 1b}
      procedure 1b definition
   {postcondition 1b}
{postcondition 0}
```

Figure 13: A subproblem hierarchy represented as a solution scope in a context. Each signature represents a subproblem, and the scope is defined from left to right.

```
linked list
   has n nodes
      each node has an integer value
```

$+\downarrow$

```
linked list
   has n nodes
      node 0 has an integer value
         integers are ordered
         possible values
            1
            2
            3
            ...
      node 1 has an integer value
         integers are ordered
         possible values
            1
            2
            3
            ...
      ...
```

Figure 15: In fact contexts, members of a conjunction are written in non-italic, members of a disjunction are written in italic. "..." means *and so on*.

We see that an order is defined upon the integers. We can use this fact to find the node with the greatest value, but it still might not be clear exactly how. It seems that we've applied + about as many times as is going to be helpful, so we now try removing facts from the scope via −.

In our most recent problem scope, we see that integers are defined, in part, by a disjunction of possible values that an integer can have. This means that an integer can have a value of either 1, or 2, or 3, and so on, but not more than one of these values at the same time. Each of the facts in this disjunction will inevitably be defined by facts of their own. For example, we could have *2 = is prime* `and` *is even*. Importantly, however, because each fact of a disjunction is mutually exclusive, such facts are not always valid. This makes them difficult to reason with, as, in the case of the integers, these facts will only be true for some values of an integer. Thus, for the moment, let us remove (via −) from the context all but one of the facts of the integer disjunction, effectively eliminating the disjunction from the context. Furthermore, let us fix $n = 3$.

We refer to this technique of disjunction elimination as **construction**, and to the smaller context that results as *a* construction.

```
linked list
   has n nodes
      node 0 has an integer value
         integers are ordered
         possible values
            1
            2
            3
            ...
      node 1 has an integer value
         integers are ordered
         possible values
            1
            2
            3
            ...
      ...
```

construction (via $-$) $\downarrow$

```
linked list
   has 3 nodes
      node 0 has an integer value
         integers are ordered
         possible values
            2
      node 1 has an integer value
         integers are ordered
         possible values
            7
      node 2 has an integer value
         integers are ordered
         possible values
            3
```

Figure 16: A construction of the original context. Construction occurs via $-$.

The definition of the integers in the resulting construction is obviously not a valid definition of the integers in the context with which we began, which means that this construction is not equivalent to the context with which we began. Given two problems $p$ and $p'$, each with identical signatures, we say

that $p$ is equivalent to $p'$ if their contexts are equivalent. Problems in different equivalence classes cannot be solved by the same solution, so a solution to our constructed problem cannot be applied to the original problem.

However, happily, our construction has no disjunctions, which makes its facts easier to reason about, and it can still be used to derive facts that can help us solve the original problem. We do this via a technique called generalization.

The construction we're left with has far fewer facts in it than the context with which we began. This higher level of abstraction means that the problem is now simple enough that we can represent it as a computer code prototype, a drawing, or a diagram, e.g.:

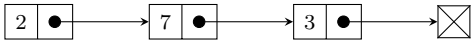


Figure 17: A visual representation of the linked list under consideration.

Such "non-abstract" representations allow us to apply techniques other than abstraction to a context (albeit indirectly). For example, examination of this diagram leads us to the observation that, if we simply traverse the list and keep track of the node with the greatest value up until that point in the traversal, we are guaranteed to end up with a pointer to the node with greatest value at the end of the traversal. This is due to the particular values of each node in the list.

```
The list l has a head and each node has a successor, which means
    that the entire list is reachable from the head of l.
⟹
{l is a linked list}
    Traverse l by beginning at the head and accessing each node's
        successor until the end of the list is reached. During
        the traversal...
    ;
    The values in the list are 2, 7, and 3, in that order.
        Reasoning in the order of a traversal of the list, 2 is
        greater than nothing, 7 is greater than 2, and 3 is not
        greater than 7.
    ⟹
    {n is a node}
        If we set the value of the head node as the current
            greatest valued node in the list, and then simply
            compare the value of the current node in the traversal
            to that of the current greatest valued node (
            reassigning the current greatest valued node if its
            value is eclipsed by the value of the current node),
            we are guaranteed to end the traversal with the node of
             greatest value.
    {the node with greatest value in l is returned}
{the node with greatest value in l is returned}
```

Figure 18: A solution with complete definitions for all actions.

At this point, we have solved the problem associated with this construction. However, since the problem associated with this construction is in a different equivalence class from the original problem, the facts which were used to derive this solution are not guaranteed to be valid in the original context. This means that, if we are to say that this is a solution to the original problem, we will need to modify its justification using facts which are guaranteed to be valid in the original context.

Our solution uses the values 2, 7, and 3 in its justification, but it really does not depend on them. Instead, it depends on the order that exists on the integers, and it turns out that we can use this fact to justify a solution for the original problem. An expression's **generality** is measured by the number of contexts in which the expression is valid, and this process of replacing members of a fact's scope with facts located in a predecessor to that scope is called **generalization**.

```
linked list
   has 3 nodes
      node 0 has an integer value
         integers are ordered
         possible values
            2
      node 1 has an integer value
         integers are ordered
         possible values
            7
      node 2 has an integer value
         integers are ordered
         possible values
            3
```

generalization (via $-$) $\downarrow$

```
linked list
   has n nodes
      node 0 has an integer value
         integers are ordered
      node 1 has an integer value
         integers are ordered
      node 2 has an integer value
         integers are ordered
      ...
```

Figure 19: Generalization occurs by noticing that the **2**, **7**, and **3** expressions can be replaced by the **integers are ordered** expression, which means that the **possible values** of each node don't matter.

Note that we could have simply created a construction for each possible sequence of integer values in the input list, derived a solution for each, and then conditionally used each solution depending on which sequence of values appears in the input. In the case of this problem, the set of integers is infinitely large, which would make this approach impossible. It is a possible approach for smaller disjunctions, but it is inelegant, because it would require a separate solution for each set of possible disjunction values, which is undesirable because it means that we would need to have an awareness of all of the facts which are required to all of justify those solutions. Thus, the **elegance** of a solution is measured by the number of facts by which it is justified.

# Conclusions

Abstraction reifies the elements that are used to solve problems and allows us to attempt to quantify both problems and solutions. It allows us to not only apply the same set of tools to many different problems, but also gives us a language with which we can discuss problems and their solutions.

However, the application of abstraction to problem solving is not without challenges. The framing of a problem statement as a context depends upon the existence of a sufficiently detailed problem statement, and it is often the case in practice that either the problem statement is not sufficiently detailed, or there is no problem statement to begin with. In these cases, consultation of henchmen or documentation may be helpful. A related issue is that, while this paper has dealt (for the most part) with unambiguous contexts, in practice, most contexts are likely to be ambiguous to varying degrees. This may be due to different engineers having different understandings of certain aspects of a problem, or perhaps to the aforementioned issue of underspecification, or perhaps to the fact that we can't know a priori whether or not a particular expression will be helpful in satisfying a goal. This means that, in practice, most applications of $+$ and $-$ are likely to be assumptions, if only to a trivial extent.

Thus, we have gone from an imperfect technique (problem solving via pattern matching) to another imperfect technique (problem solving via abstraction). We can only hope that the latter is less imperfect than the former!

# References


[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[Gri87] David Gries. *The Science of Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1987.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[Man89] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented Modeling and Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[RGI75] D. T. Ross, J. B. Goodenough, and C. A. Irvine. Software engineering: Process, principles, and goals. *Computer*, 8(5):17–27, May 1975.

[SM09] Alexander A Stepanov and Paul McJones. *Elements of programming*. Addison-Wesley Professional, 2009.

[WF67] Robert W. Floyd. Assigning meanings to programs. 19, 01 1967.

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, April 1971.