DOI:10.1145/3490685

TURING LECTURE

Jeffrey D. Ullman and Alfred V. Aho are recipients of the 2020 ACM A.M. Turing award. They were recognized for creating fundamental algorithms and theory underlying programming language implementation and for synthesizing these results and those of others in their highly influential books, which educated generations of computer scientists.

BY ALFRED AHO AND JEFFREY ULLMAN

Abstractions, Their Algorithms, and Their Compilers

1. Abstractions

COMPUTATIONAL THINKING, WHICH centers around devising abstractions for problems so they can be solved with computational steps and efficient algorithms, is a concept that serves not only computer science (CS) but progressively more of science and everyday life.

After Jeannette Wing published her influential paper on computational thinking,[50] the discussion of computational thinking has greatly expanded in scale to include topics such as modeling natural processes as information processing.[18]

At the heart of computational thinking is abstraction, the primary subject of this essay. While abstractions have always been an essential part of computer science, the modern emphasis on computational thinking highlights their importance when teaching CS to a broad audience.

Abstractions play an important role in shaping virtually every field of science. However, in computer science, abstractions are not tethered to a physical reality, so we find useful abstractions pervading the field. For example, we shall encounter in Section 4 a vital abstraction from physics: quantum mechanics. There is a derived computing abstraction called quantum circuits, which starts with the physical ideas, but has led to programming languages for its simulation, to theoretical algorithms that exploit its unique capabilities and that

one day may be implemented on a large-scale machine.

Every abstraction in computer science consists of the following:

1. A *data model*, which is one or more types of data and possibly relationships among them. For instance, an undirected graph could be described as consisting of a set of nodes and a set of edges; each edge is a set of two nodes.
2. A way of manipulating that data—that is, a programming language of some sort. This might be a conventional programming language, or it might be as simple as a few specific operations. This language always has a formal semantics—a specification for how programs affect the data.

Notice it is the second part that distinguishes abstractions in computer science from abstractions in other fields.

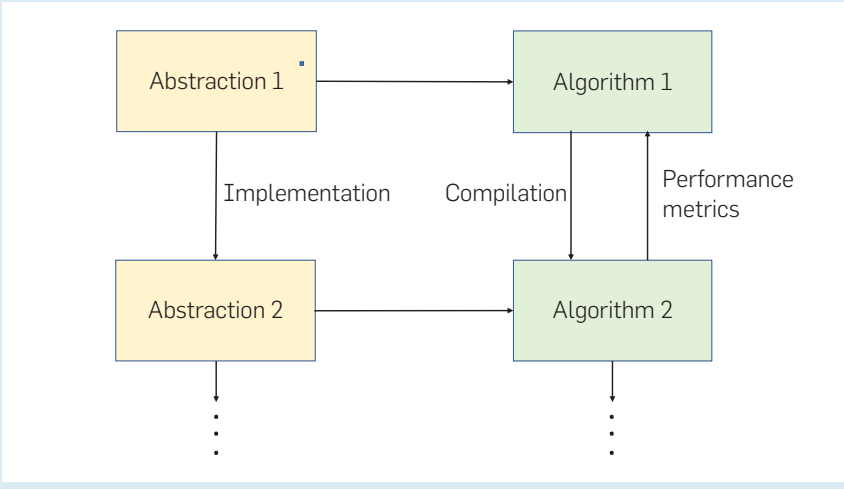
Each abstraction thus allows us to design algorithms to manipulate data in certain specific ways. We want to design "good" abstractions, where the goodness of an abstraction is multidimensional. The ease with which an abstraction can be used to design solutions is one important metric. For example, we shall discuss in Section 3.1 how the relational model led to the proliferation in the use of databases. There are other performance metrics, such as the *running time*, on serial or parallel machines, of the resulting algorithms. Likewise, we favor abstractions that are easily implemented and that make it easy to create solutions to important problems. Finally, some abstractions offer a simple way to measure the efficiency of an algorithm (as we can find "big-oh" estimates of the running time of programs in a conventional programming language), while other abstractions require that we specify an implementation at a lower level before we can discuss algorithm efficiency, even approximately.

1.1. Compilation. Some abstractions are at a sufficiently high level that they do not offer meaningful performance metrics. Thus, the operations of a high-level abstraction may need to be implemented at a lower level—a level closer to what we think of as program execution. Indeed, there may be several levels of abstraction at levels that are progressively closer to the machine itself. As suggested in Figure 1, the operations of a high-level abstraction (Abstraction 1) may be implemented by a lower-level abstraction (Abstraction 2), which in turn may be implemented by still-lower-level abstractions (not shown). There are interesting hierarchies of abstraction that take us from high-level programs to machine instructions, to physical hardware, to logical gates, to transistors, and finally to electrons. We shall, however, concentrate only on the higher levels.

An algorithm using the operations of Abstraction 1 is compiled into an algorithm in the lower-level Abstraction 2. In this paper, we shall use the term *compiler* in a very general sense—not just the conventional compiler of programming languages as was the focus of the "Dragon book"[9] but any algorithm for turning programs of one abstraction into programs of a second, presumably lower-level abstraction. Thus, in some cases, the compilation process is straightforward, with each operation at the higher level replaced by one or more specific operations at the lower level. In other cases, especially when the compilation is from a conventional language (think C) to a machine-level language, the translation algorithm is very complex. In still other situations, such as when the high-level abstraction uses powerful algebraic operations such as linear algebra or relational algebra, optimization is critical, since naive compilations often result in algorithms that take orders of magnitude more time than those generated by optimizing compilations.

It may be that Abstraction 2 is sufficiently close to the machine that it has meaningful performance metrics. If so, Abstraction 1 can inherit those metrics to provide a notion of quality for algorithms written in Abstraction 1. However, we should remember that the high-level abstraction usually can be implemented in several different lower-level abstractions. Each of these abstractions may provide a radically different notion of running time or other metrics and thus imply different notions of algorithm goodness at the high level.

Figure 1. Layers of abstractions and algorithms.



EXAMPLE 1.1. *For an example that should be familiar to many readers, and which we shall discuss in detail in Section 2.1, Abstraction 1 could be regular expressions (REs), whose data model is sets of strings and whose "programming language" is the regular expressions that can be formed from the three operations: union, concatenation, and Kleene star. We can implement regular expressions by converting them to deterministic finite automata (DFA), which thus play the role of Abstraction 2. An algorithm in the RE abstraction is the RE itself. That algorithm can be "compiled" into a DFA, which in turn is "compiled" (that is, simulated) in some conventional programming language, which in turn is compiled into the language of the machine. Given*

a reasonable simulation, the DFA running time is O(1) per input symbol. Thus, the running time for the original RE is also O(1) per input symbol, even though that point is far from obvious given only the definition of REs.

1.2. A taxonomy for abstractions. We can identify at least four different kinds of abstractions that are distinguished by their intended purpose. In the discussions that form the body of this paper, we shall give examples of each and explore their interactions.

1.2.1. Fundamental abstractions. Like all abstractions, these consist of a data model and operations. These abstractions are often thought of as *abstract data types*[31,46] as implemented in object-oriented programming languages. However, fundamental abstractions do not have a specific implementation for the operations or a specific data structure to represent their data. One can also liken these abstractions to *interfaces* as in Java,[48] but unlike the interface, these abstractions have an intended meaning for their operations, not just a name for the operations.

There are actually two somewhat distinct purposes for studying fundamental abstractions. In some cases, they represent common operations that are worth studying in their own right and which have a variety of approaches to implementation. For instance, we discuss the dictionary (a set with operations insert, delete, and lookup) in Section 1.4. Other examples of this type include stacks, queues, priority queues, and a number of other abstractions that we have cataloged in Aho et al.[7] and Aho and Ullman.[10]

Other abstractions are extensive enough to support large components of application programs. Familiar examples include trees and graphs of various kinds—for instance, directed, undirected, labeled, and unlabeled. We shall discuss one important example—flow graphs—in Section 2.3.

These abstractions have an extensive set of operations that can be combined in various ways. However, the operations are not *Turing-complete* (capable of describing any computation that can be performed by a Turing machine) by themselves. Rather, they are presumed to be embedded in a Turing-complete language, in which algorithms using this model are constructed. For example, in a graph abstraction, we might have an operation such as "find adjacent nodes." Outside this abstraction, we might suppose that there is a programming language that allows iteration over all the adjacent nodes. The implementation of this operation and the representation of the graph itself are left unspecified, so we have no concrete notion of running time. We can draw an analogy between these abstractions and the typical class and its methods in an object-oriented programming language. The distinction is that the methods for a class have specific implementations in the underlying programming language. We could similarly regard things such as programming-language libraries or TeX packages as abstractions of this type.

1.2.2. Abstract implementations. These represent approaches to implementation, presumably implementation of one or more fundamental abstractions. These are not themselves Turing-complete languages, and they typically can be compiled into several different machine models—for example, serial or parallel machines, or models that assume main or secondary memory. Each of these machine models provides a notion of running time, which can be translated into a running time for the abstract implementation and then into running time for the supported fundamental abstraction. For example, we shall discuss the hash table as an implementation of the dictionary in Section 1.6. Many other data structures fall into this category, such as linked lists or trees of various types (see Sections 1.5 and 1.7.2). Also in this group are automata, such as deterministic or nondeterministic finite automata (see Sections 2.1.1 and 2.1.3) and shift-reduce parsers (see Section 2.2.1).

1.2.3. Declarative abstractions. One of the most important uses of abstractions is to foster a style of programming where you say what you want done, but not how to do it. Thus, we find many different abstractions that consist of a data model and a programming language that is at a higher level than that of conventional languages; often these languages are algebras of some sort. Examples are regular expressions, to be discussed in Section 2.1, and relational algebra, which we mention in Section 3. Context-free grammars (Section 2.2) are another example of this type of abstraction, although not strictly algebraic.

The special characteristic of these abstractions is that their compilation requires a serious degree of optimization. Unlike optimization of conventional languages, where you are happy to speed up running time on one machine by a factor of two, for these abstractions there can be orders of magnitude difference between the running times of good and bad implementations. Another characteristic is that the programming language for a declarative abstraction is not Turing-complete. Undecidability properties of any Turing-complete language would preclude the existence of an optimizer that deals effectively and generally with what a program wants to do without being told how to do it.

1.2.4. Computational abstractions. In contrast to the abstract implementations, these abstractions are close to a physically implemented machine. That is, no one would build a machine for the sole purpose of implementing an abstract implementation, but one normally implements a computational abstraction or something to which it is easily translated. Thus, computational abstractions offer meaningful performance metrics, even if they are not 100% accurate.

You are probably familiar with a number of these abstractions, since they include all the common programming languages as well as machine instruction sets. Other abstractions of this type are more theoretical, such as the random-access machine (RAM) model[19] or the parallel-RAM (PRAM) model.[20] Here, we shall talk in Section 1.7 of a model of a conventional machine that emphasizes the role of secondary storage. We shall also discuss abstractions for parallel computation: bulk-synchronous in Section 3.5 and MapReduce in Section 3.6.

While many computational abstractions relate to conventional computers, there are some exceptions. The Turing machine[43] is one, and there are others that are not even Turing-complete, yet play an important role in computer

science. For instance, following Claude Shannon's MS thesis, Boolean circuits and Boolean algebra are among the earliest abstractions used in the developing science of computing. And the quantum circuit abstraction, which we discuss in Section 4, is among the newest.

1.3. Some taxonomic comments. We do not claim that this taxonomy is exhaustive or that it is the best possible taxonomy. The reader is invited to add to or rethink this structure. However, we do believe that this organization reflects four of the most important developments in the field of computer science. Fundamental abstractions are, in a sense, dictated to us by the requirements of applications. Abstract implementations, on the other hand, represent the response of computer scientists to support the fundamental abstractions, and they represent a lot of the earliest thinking about data structures. Declarative abstractions represent the tendency of computer science to make programming progressively easier by supporting notations for encapsulating large, complex operations as single steps. And computational abstractions represent the influence of changing hardware technology on how we compute.

1.4. An exploration of the abstraction space. To gain some perspective on the nature of abstraction chains and their relationships, we shall look at a common example of a fundamental abstraction: the dictionary. We then explore the implementation levels below that and their effect on running time evaluations.

Example 1.2. *The* dictionary *is a common example of an abstraction that has many alternative implementations and illustrates some of the issues that crop up as high-level abstractions are compiled down into lower-level ones. The data model for a dictionary consists of the following:*

1. *A universal set U.*
2. *A subset S of U. Initially, S is empty.*

The "programming language" for the dictionary consists of straight-line sequences of the following three operations:

1. `insert(x)` *inserts element x of U into the set S if it is not already there; that is,* $S := S \cup \{x\}$.
2. `delete(x)` *removes x from S;* $S := S - \{x\}$.
3. `lookup(x)` *returns* `true` *if x is in S and returns* `false` *otherwise.*

For instance, a dictionary can be used to describe the behavior of a symbol table in a compiler. U would be the set of possible identifiers of the programming language. As the compiler scans the program, S will be the set of identifiers that have defined meaning at each point in the program. However, for a symbol table, you need to attach data to each identifier—for example, its defined data type and the level of nested blocks in which it appears (so we can distinguish identifiers with the same name). When the compiler finds a declaration, it inserts the declared identifier into the set S. When it reaches the end of a procedure or function, it deletes the identifiers associated with that program block. When an identifier is used in the program, the compiler looks up that identifier and retrieves its type and other necessary information.

Notice that the programming language for the dictionary is rather trivial, and it certainly does not have the power of a Turing machine. Moreover, there is no real notion of algorithm design, since the "programs" are simply a reflection of what some other process is doing—for instance, the symbol table operations mentioned in Example 1.2. Likewise, there is no real notion of running time, since it is unclear how long each operation takes. We could define each operation to take unit time, but since we cannot control the length of the "program," there is no meaning to this running time.

1.5. Implementations of the dictionary. Many different abstract implementations could be used to implement a dictionary. Linked lists will serve, although they do not offer a good running time, even for the most efficient implementation of lists.

Example 1.3. *The linked list is an abstract implementation with which all should be familiar. Its data model consists of the following*:

1. Cells *containing a* value *(member of some universal set U) and a* link *to another cell (possibly null).*
2. Headers, *which are simply named links (possibly null) to a cell.*

We shall assume the reader is familiar with the typical operations allowed, such as creating cells or headers, inserting and deleting cells from lists, and returning the data contained in a designated cell. You can implement a dictionary by making a linked list of all the elements in the set S. Compilation of the three dictionary operations into list operations is straightforward.

If we assume the linked list is implemented in the RAM model of a computer, we have a realistic notion of running time. We can assign one time unit to each of the basic operations on the cells of a list, as we know that on a RAM, each of these operations will take a constant amount of time, independent of the size of the set S. That observation lets us lift the RAM notion of running time to the linked list notion of running time and then to the level of the dictionary. Unfortunately, the news is not good. On average, we have to go at least halfway down the list, often all the way to the end, to implement any of the dictionary operations. Thus, the running time for a single dictionary operation is proportional to the size of the set S at the time.

Another well-understood class of abstractions which implements a dictionary uses search trees. When the algorithms for the three dictionary operations keep the tree balanced—for instance, AVL trees[2] or red-black trees[22]—the running time for each operation is logarithmic in the size of the set S at the time of the operation. But the normally preferred abstraction for implementing the dictionary is the hash table,[36] which we shall consider next.

1.6. The Hash abstraction. The data model for Hash consists of the following:

1. A universal set U.
2. A number of *buckets* B; the buckets are numbered from 0 to $B - 1$.
3. A *hash function* h from U to $\{0,1,\ldots,B-1\}$. Each bucket b is a subset of those elements x of U such that $h(x) = b$.

The usual operations are to compute

$h(x)$—where x is a member of U—and to insert, delete, or lookup x in the bucket numbered $h(x)$. For instance, the insert operation for the hash table will be denoted `h-insert(x,b)`, where $b = h(x)$. Programs for hash consist of alternatingly computing some $h(x)$ and then doing one of the three operations on x and the bucket $h(x)$.

Compiling a dictionary program into a hash program is simple. For example, the dictionary operation `insert(x)` is translated into `b:=h(x); h-insert(x,b)`.

Hash is not sufficiently close to the machine that we can use it directly to determine running time. One problem is that hashing is rather unusual, in that the worst-case performance, where all elements of the set S wind up in the same bucket, is wildly worse than the average case, when we average over all possible hash functions. To make things simple, we shall correctly assume that in the average case, almost all buckets contain close to the average number of elements—that is, $\|S\|/B$. But even after we agree to talk only of the average case, we still do not know how long each operation on an element and a bucket takes.

Essentially, each bucket is a little dictionary by itself, so we have to decide how to implement its operations. If the size of S remains on the order of B, we can use a linked list implementation of buckets and expect that each operation takes $O(1)$ average time on a RAM or a real machine. If, however, $\|S\|$ is much larger than B, then the average length of the list that represents a bucket is $O(\|S\|/B)$. That is still better than $O(\|S\|)$ per operation as would be the case for a linked list. However, when S is so large it cannot fit in main memory, the RAM model no longer applies, and we need to consider another computational abstraction entirely.

1.7. The secondary storage abstraction. As an alternative to the RAM computational abstraction, where any piece of data can be accessed in $O(1)$ time, we can introduce locality of access at several levels. We shall discuss only an abstraction that has disk-based secondary memory, where large blocks of data (think 64KB) are moved as a whole between disk and the main memory. Data must be in main memory if it is to be read or written. The cost of moving a block between main and secondary memory is very large compared with the cost of the typical operations that would be done on the data itself when it is in main memory. Thus, it becomes reasonable to regard running time in this new model as simply the number of disk I/Os—that is, the number of times a block is moved from secondary to main memory or vice versa.[a]

In a secondary storage model of the underlying machine, the best way to implement a hash table is somewhat different from the preferred approach using a RAM model. In particular, each bucket will consist of one or more entire disk blocks. To take advantage of locality, we want the typical bucket to consist of as few disk blocks as possible, but we want to make those disk blocks as full as we can. Thus, suppose main memory is capable of holding M elements of the universal set, while disk blocks hold P such elements. Then we want B, the number of buckets, to be $B = M/P$, so we can hold one disk block for each bucket in main memory, and that block will likely be close to full.

As the size of the set S grows, we use a linked list of disk blocks to represent each bucket, only the first of which is in main memory. The three dictionary operations at worst require us to examine all the disk blocks in a single bucket. So, on average, we expect the number of disk I/Os per operation to be $O(\|S\|/BP)$, since the elements of S will be divided approximately evenly among B buckets, and the elements of a bucket will be packed P to a disk block. Since $B = M/P$, the running time per operation is $O\|S\|/M)$.

1.7.1. Database operations via hashing. Is this running time impressive? Not really. Had we used a linked list implementation but allowed the lists to grow randomly onto disk, we could have used M buckets, and the average bucket would be represented by a linked list of length $\|S\|/M$. Even if each cell on the list were in a different disk block, we would still need only $O(\|S\|/M)$ disk I/Os per dictionary operation. However, there is an important class of operations from the world of database management where the disk block model yields important efficiencies. In these operations, of which hash-join[25] is a major example, we have only to divide a set S into buckets, while neither deleting nor looking up elements.

EXAMPLE 1.4. *Let us consider a simpler problem than hash-join, but one that has a similar solution in the disk-based abstraction. Suppose we are given a set S of pairs* (x, y), *and we want to* aggregate *on x by computing a table that for each x gives the sum of the associated y's. We assume S is so large that it does not fit in main memory. As above, we shall use B = M/P buckets, but our hash function is only a function of x, not y. We then insert each member of S into the proper bucket. Initially, the block for each bucket that is in main memory is empty. When it fills up, we move it to disk and create a new, empty block in main memory for that bucket. The new block links to the block we just moved to disk, which in turn links to prior blocks for that bucket, if any. The number of disk I/Os required by the entire partition of S will be approximately the number of blocks needed to hold S—that is,* $\|S\|/P$. *That is the minimum possible number of disk I/Os needed to process a set of that size.*

Then, we process the buckets one at a time. Assuming that the buckets themselves are small enough to fit in main memory—that is, $\|S\| \leq M^2/P$*—we only need to move each disk block once into main memory. Since a bucket contains either all or none of the pairs* (x, y) *with a given x, we can sort each bucket in turn, on the values of x and thus compute the sum of y's for each of the x's that hash to that bucket. Since in this model we only count disk I/Os, and in fact the time to sort a bucket is often less than the cost of moving its blocks into or out of main memory, the running time of this algorithm, which is* $O(\|S\|/P)$ *disk I/Os, is realistic and, to within a small constant factor, the best imaginable.*

1.7.2. B-trees. Another interesting consequence of the secondary storage abstraction is that it provides an implementation of the dictionary

a While we shall not talk about it, there is another, similar abstraction where the data is assumed in main memory, but needs to be moved into a cache to be read or written. Data is moved between main memory and cache in units of *cache lines*, which are typically about 1/1000th the size of disk blocks, but still make locality of access important.

abstraction that is generally more efficient than hashing. In a RAM model of computation, even a balanced search tree is not generally considered more efficient than a hash table for implementing the dictionary. But in the secondary storage model, the analog of a balanced search tree is really quite efficient. A *B-tree*[13] is essentially a balanced search tree, where the nodes of the tree are entire disk blocks. Each leaf block is between half and entirely full of members of the set S being represented. Further, each interior node is at least half-full of values that separate the contents of the subtrees for its children and pointers to those children.

Since disk blocks are large, each interior node has many children, so much so that in practice even very large sets can be stored in a three-level B-tree. Moreover, at least the root, and perhaps the children of the root, can all be stored in main memory. The result is that, excluding occasional disk I/Os for rebalancing the tree, only two to four I/Os are needed for each insert or delete operation and only one to two I/Os for a lookup.

2. Abstractions for Compilers

Modern compilers refine the translation process into a composition of phases, where each phase translates one representation of the source program into another semantically equivalent representation, usually at a lower level of abstraction. The phases in a compiler typically include lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and target code generation. A symbol table that is shared by all the phases is used to collect and make available information about the various constructs in the source program. The first four phases are often called the *front end* of the compiler, and the last two the *back end*.

The story of advances in compiler implementation involves a number of important abstractions. We shall talk specifically about three such abstractions: regular expressions, context-free grammars, and flow graphs. The first two are declarative abstractions with interesting optimization stories. The third, while not declarative, also presents interesting implementation challenges.

2.1. Regular expressions and lexical analysis. Lexical analysis, the first phase of the compiler, reads the source program as a sequence of characters and maps it into a sequence of symbols, called tokens, which is passed on to the next phase, the syntax analyzer.

Example 2.1. *If the source program contained the statement*

```
fahrenheit = centigrade * 1.8
    + 32
```

the lexical analyzer might map this statement into the sequence of seven tokens:

```
<id> <=> <id> <*> <real> <+> <int>
```

Here, `id` *is the token for any program variable, or* identifier. *The operators* `=`, `*`, *and* `+` *are tokens by themselves, and the two constants are turned into tokens* `real` *and* `int`, *respectively.*[b]

One great advance in compiler construction was the creation of a *lexical-analyzer generator*—a program such as Lex[30] that takes as input the description of the tokens and generates a program that breaks the source program into tokens and returns the sequence of tokens that corresponds to the source program. The abstraction that made Lex possible is the *regular expression*.[26] As originally defined by Kleene, we can think of regular expressions as an abstraction whose data model is strings over some specific alphabet of characters—for example, ASCII—and whose programming language is an algebra with operators union, concatenation, and the Kleene star, or "any number of" operator. We assume the reader is familiar with these operators and their effect on sets of character strings. In practice, systems such as Lex, which use the regular-expression abstraction, employ a number of useful shorthands that make it easier to write regular expressions but do not change the sets of strings that can be defined in this abstraction.

Example 2.2. *The sets of strings that are legal identifiers in some programming languages might be defined as follows:*

```
letter = [a-zA-Z]
digit = [0-9]
id = letter(letter+digit)*
```

In this shorthand, an expression like `a-z` *stands for the union of the one-character strings with ASCII codes between* `a` *and* `z`. *So the regular expression for* `letter` *is, in the original set of three operators:*

$$a + b + \cdots + z + A + B + \cdots + Z$$

Digits are defined analogously, and then the set of strings for the token `<id>` *is defined to be those strings consisting of any letter followed by any string of zero or more letters and/or digits.*

2.1.1. Before Lex: bibliographic search. It was well understood from theoretical studies that the regular expression abstraction could be compiled into one of several abstract implementations, such as deterministic or nondeterministic finite automata (NFA and DFA, respectively). However, there were still some techniques to be discovered when practical problems needed a solution. One interesting step was taken at Bell Laboratories in connection with the first attempt at automated search for relevant literature. They had on tape the titles of the entire Bell Labs library, and they had developed software to take a list of keywords and find the documents with those keywords. However, when given a long list of keywords, the search was slow because it made one pass over the tape for each keyword.

A big improvement was made in Aho and Corasick.[6] Instead of searching for each keyword separately, the list of keywords was treated as a regular expression for the set of all strings that contained any occurrence of a keyword, that is:

$$\cdot * \left(w_1 + w_2 + \cdots + w_n\right)$$

Note that the dot is an extension that stands for "any character." This

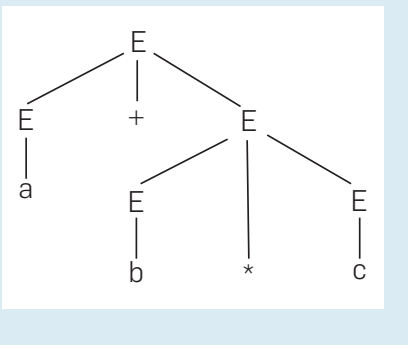
b In practice, the lexical analyzer would produce not only a sequence of tokens but would include with certain tokens a value that distinguished one instance of a token from another. For example, each `id` token would have an associated pointer to the symbol table, and tokens such as `real` or `int` would have pointers to a table that gave the actual values of those constants. Operator tokens such as `=` would not have an associated value. We shall ignore these associated values in the discussion that follows.

expression was converted to a deterministic finite automaton. It was then possible to make a single pass over the tape, regardless of how many keywords were involved. Each title was examined once by the finite automaton to see if any of the keywords were found therein.

2.1.2. Design of a lexical-analyzer generator. In essence, a lexical-analyzer generator such as Lex uses the same idea as in Section 2.1.1. We write regular expressions for each of the tokens and then apply the union operator to these expressions. That expression is converted to a deterministic finite automaton, which is set to work starting at some point in the program. This automaton will read characters until it finds a prefix of the string that matches a token. It then removes the characters read from its input, adds that token to its output stream, and repeats the process.

There are a few additional considerations, since unlike keywords, there can be some complex interactions among tokens. For example, `while` looks like an identifier, but it is really a keyword used for control flow in a program. Thus, when seeing this sequence of characters, the lexical analyzer must return the token `<while>`, not `<id>`. In Lex, the order in which the regular expressions are listed in its input file breaks ambiguities such as this, so all you have to do is list your keywords before identifiers to make sure the keywords are treated as such, and not as identifiers. Another problem is that some tokens can be prefixes of another. We would not want to recognize `<` as a token if the next character in the input was `=`. Rather, we want to recognize `<=` as a token. To avoid such errors, the lexical analyzer is designed to keep reading as long as what it has seen so far is accepted by the finite automaton as a legal token.

Figure 2. Syntax tree for the expression a + b * c.



2.1.3. Lazy evaluation of the DFA. There is one more optimization that can improve the running time of algorithms using the regular expression abstraction: lazy evaluation. You may be familiar with the standard way of converting a regular expression to a deterministic finite automaton. The regular expression is first converted to a nondeterministic finite automaton by the algorithm of McNaughton and Yamada.[34] This conversion is simple and yields an NFA with at most $2n$ states if the regular expression has length n. Troubles start when you convert the NFA to a DFA, which requires the subset construction of Rabin and Scott.[38] In the worst case, that construction can convert the NFA with $2n$ states into a DFA with 2^{2n} states, which is not really usable, even for relatively short regular expressions. In practice, the worst case is rare—for example, the DFA constructed for lexical analysis of a typical programming language might not even have more states than the sum of the lengths of the regular expressions for each of its tokens.

However, there are other applications of regular expressions where something close to the worst case can and does occur. One of the earliest UNIX commands was *grep*, standing for "get regular expression and print." This command would take a character string and determine whether it had a substring in the language of a given regular expression. The simplest implementation would convert the regular expression to an NFA and then to a DFA and let the DFA read the string. When the DFA was large, much more time was spent converting the NFA to a DFA than was spent scanning the string.

However, when the regular expression is to be used only once to scan a string, there are more efficient ways to implement a command such as `grep`. Ken Thompson's first research paper[42] showed that instead of converting a small NFA to a big DFA, it was more efficient to directly simulate the NFA. That is, an NFA reading a string is typically in a set of states after reading each character. So, just keep track of those NFA states after each character and, when reading the next character, build the set of states reachable on that character from the previous set of states.

Even more efficiency can be achieved by lazy conversion of the NFA to a DFA. That is, read the input string one character at a time, and as you go, tabulate the sets of NFA states that are actually produced by the prefix read so far. These sets of NFA states correspond to DFA states, so we build only that part of the DFA's transition table that is needed to process this particular input string. If the DFA for the given regular expression is not too large, most or all of it will be constructed before we finish processing the string, and we get the benefit of using the DFA directly, rather than constructing sets of NFA states after each character of the string. But if the DFA is larger than the string, we never construct most of the DFA, so we get the best of both cases. This improvement was implemented in a version of `grep` called `egrep`.[4]

2.2. Context-free grammars and parsing. The second phase of the compiler, the syntax analyzer or "parser," maps the sequence of tokens produced by the lexical analyzer into a tree-like representation that makes explicit the grammatical structure in the sequence of tokens. A typical representation is a syntax tree, in which each interior node represents some structure and the children of that node represent the components of that structure.

EXAMPLE 2.3. *For example, a syntax analyzer might map the sequence of tokens*

```
a + b * c
```

*into the syntax tree shown in Figure 2. Here, E represents an expression. The operands a, b, and c are expressions by themselves. But b * c is also an expression, composed of the operator token * and the two expressions, b and c. At the root, we see yet another expression, this one using the operator + and the two operand expressions, a and b * c.*

It is important to observe one of the many conventions regarding operator precedence. *Normally, multiplication takes precedence over addition, which is why the syntax tree multiplies b times c before adding a, rather than adding a and b first.*

The desired structure of the syntax trees for a given programming language is normally defined by a declarative abstraction, the *context-free grammar* (CFG), a concept with which we expect you are familiar. A CFG is a collection of rules called *productions* that provide ways in which various syntactic categories, such as expressions or statements, can be constructed from other syntactic categories and *terminals*, which are the tokens produced by the lexical analyzer. For instance, if E represents the syntactic category of well-formed expressions of the language, then we might find rules such as

$$E \rightarrow E + E$$

meaning that one way to construct an expression is to put a plus sign between two smaller expressions.

2.2.1. LR(k) parsing. In the 1960's, there was a series of proposals made regarding how to construct efficient syntax analyzers from CFGs. It was recognized that, for common programming languages, it was possible to make a single left-to-right scan of a program, without backtracking, and to build the syntax tree according to the grammar for that language, provided the grammar had certain properties.

Some decisions were tricky. For example, when processing the expression $a + b * c$, after reading only $a + b$ you must decide whether to combine the expressions a and b with the plus sign to make a larger expression. If you look ahead one token and see the $*$, you know that it is not correct to combine a and b, but rather you must proceed further and eventually combine b with c. Only at that point is it correct to combine a with the expression $b * c$.

This style of syntax analysis is called *shift-reduce parsing*. You keep a stack of symbols, which may be tokens or syntactic categories. As you scan the input, you decide at each step whether to *shift* the next input token by pushing it onto the stack or to *reduce* the symbols at the top of the stack. When you reduce, the symbols reduced must be the right side of some production of the CFG. These symbols are popped off the stack and replaced by the left side of the same production. In addition, you create a node of the syntax tree for the symbol on the left side of the production. Its children are the roots of the trees corresponding to the symbols just popped off the stack. If a token is popped from the stack, its tree is just a single node, but if a syntactic category is popped, then its tree is whatever had been constructed previously for that symbol on the stack.

After a series of increasingly more general proposals, summarized in Aho et al.,[9] Don Knuth proposed LR(k) parsing,[27] which in essence works on the most general class of grammars that can be parsed correctly on a single left-to-right scan of the input, using the shift-reduce paradigm and looking at most k symbols ahead on the input. This work seemed to settle the matter of how syntax analyzers should be constructed. However, not every CFG, or even every CFG for a typical programming language, satisfies the conditions necessary to be an LR(k) grammar for any k. While it appeared that common programming languages did have LR(1) grammars—that is, grammars that can be shift-reduce parsed using only one symbol of lookahead on the input—these grammars were quite complex to design and often had an order of magnitude more syntactic categories than were needed intuitively. For example, instead of one syntactic category for all expressions, you need one syntactic category for each level of operator precedence in the language, of which there are typically a dozen or more levels.

2.2.2. The Parser-Generator Yacc. Thus, following Knuth's paper, there were several attempts to find ways to use the LR(1) parsing approach but to make it work on simpler CFGs. We were inspired by one of our fellow graduate students at Princeton, Al Korenjak, whose thesis was on ways to compress LR(1) parsers.[28] It occurred to us that, for common languages, one could start with a grammar that was not LR(1) and still build a left-to-right shift-reduce parser for the grammar. When a grammar is not LR(1), there will be some situations where the grammar tells us to reduce and shift or to reduce using two different productions. But we could resolve the ambiguity in practical cases by considering precedence of operators and looking one token ahead on the input.

Example 2.4. *Consider the situation suggested in Example 2.3. After processing the $a + b$ of input $a + b * c$, we would have at the top of the stack $E + E$, where both a and b had previously been reduced to expressions. There is a production $E \rightarrow E + E$, so we could reduce $E + E$ to an E and build a node of the parse tree with label E and children E, +, and E. But the fact that $*$ takes precedence over +, and we see $*$ as the next input symbol, tells us that it is instead correct to shift $*$ onto the stack. Later, we shift the c as well and reduce the c to an expression E. At this point we have $E + E * E$ at the top of the stack. We correctly reduce the top three symbols to an E, leaving $E + E$. Now, it is correct to reduce these symbols to an E because there is nothing left on the input (or there is something else on the input that is not part of an expression, such as the semicolon that ends a statement). In this manner, we would produce the syntax tree that was shown in Figure 2.*

Steve Johnson, our colleague at Bell Labs, took this idea and implemented a parser-generator called Yacc.[8] To help resolve ambiguities between shift and reduce actions, or between reduction by two different productions, Yacc uses the order in which productions appear. Whichever production appears first is preferred in situations where two productions could each be used to reduce. To resolve conflicts between shift and reduce, whichever operator appears first in the Yacc input file is assumed to take precedence.[c]

Yacc quickly became an essential tool for implementing compilers—not only compilers for conventional programming languages but for the many "little languages" that have a more limited purpose. Together with Lex, Yacc provides an easy way to experiment with the design of the syntactic structure of a new language. The two tools are often used in one-semester compiler courses in academia, in which students design and implement a new domain-specific programming language.[5]

c Although lexical analysis precedes parsing in a compiler, historically, Yacc preceded Lex. Thus, the trick of exploiting order of appearance to resolve ambiguities really belongs to Steve Johnson.

2.3. Flow graphs and code optimization. A compiler's code optimization phase takes an intermediate form of the source program, typically something like *three-address code*. There, large expressions are broken into simple steps where (at most) two operands have a single operator applied to yield a single result—`a := b+c`, for example. Three-address code uses other steps, such as branching based on a single value, but we shall ignore these in this brief discussion.

A key abstraction for code optimization is the *flow graph*. Originating with Vyssotsky and Wegner,[45] the scientific study of algorithms for this abstraction is generally regarded as beginning with the work of Allen.[11] The data model for a flow graph begins with a directed graph, where each node represents a *basic block* of the intermediate code—a sequence of three-address steps that can only be executed without branches in the middle. There is more to the data model which we'll get to in a moment. Algorithms using the flow-graph abstraction are normally intended to gather information about the program as a whole and identify when certain optimizations are possible.

EXAMPLE 2.5. *One useful kind of information we can gather using the flow-graph abstraction is* reaching definitions. *We want to know, at each point p in the intermediate-language program and for each variable X of that program, where in the program might X last have been defined. Typically, there will be several different places. But if there is only one such place, and that place assigns a constant c to X, then we know that at point p, X definitely has the value c. If X is used at p, then we can substitute c for the use of X, which usually results in code that can be executed faster than if we have to read the value of X.*

There is an elegant flow-graph abstraction due to Kildall.[24] In this framework, there is a semilattice L of values, which is part of the data model, along with the graph itself. Associated with each node of the flow graph are two members of L, one for the beginning and one for the end of the three-address code that the node represents. The operations for this abstraction are as follows:

1. A *transfer function* f_n associated with each node n. The purpose of f_n is to compute the value $O = f_n(I)$ associated with the end of node n from the value I associated with the beginning of n. For instance, if we are trying to compute reaching definitions, as in Example 2.5, then f_n would compute O by throwing out of I every point that defines a variable that is also defined by the three-address code that n represents. And if there are one or more definitions of a variable X in n, then f_n would add in the last of these definitions of X.
2. A *confluence operator* that allows us to compute the value associated with the beginning of a node from the values at the ends of all its predecessors in the flow graph. For instance, if reaching definitions were being computed, then we would use union as the confluence operator, because for a definition of a variable X to reach the beginning of a node of the flow graph, it is only necessary for that definition to reach the end of one of its predecessors. There are some algebraic requirements on the confluence operator, namely those of the meet operator of a semilattice: commutativity, associativity, and idempotence.

The flow-graph abstraction just described allows for many different algorithms to be used to solve for the values at the beginning and ends of each node, given the transfer functions associated with each node and the confluence operator. Most of these approaches are iterative in nature, converging slowly to the correct answer. However, there are a number of interesting variants, often starting from the *Interval* approach of Allen,[11] which takes advantage of the common structure of programs, where loops are nested only to a small depth. Moreover, these algorithms can be used to solve not only for reaching definitions but also for essentially any of the types of information about programs that are useful for code optimization. You are invited to examine the extensive story in Aho et al.[9]

3. Large-Scale Data Abstractions

When we think about the largest available datasets and the algorithms that can be used to manipulate them, we need several new abstractions. The secondary storage model of Section 1.7 is important, but there are other abstractions that express various forms of parallelism and distributed computing. We shall outline the most relevant abstractions here.

3.1. The relational model of data.

To begin, the *relational model* of Codd[15] has proven to be central to the processing of large-scale data. In brief, data is organized as a collection of tables, or relations, two examples of which are shown in Figure 3. On the left is a relation called Cities, which has two columns: City and State. The *schema* of a relation is its name and the list of column names—in this case, Cities (City, State). The relation itself is a set of rows or *tuples*. For example, one of the rows of the relation Cities is (Toronto,

Figure 3. Two relations: Cities (City, State) and States (State, Country, Pop).

City	State
Toronto	Ontario
New York	New York
Mumbai	Maharashtra
Montreal	Quebec
San Jose	California
Chennai	Tamil Nadu

Cities

State	Country	Pop
California	USA	38.51
Maharashtra	India	114.2
New York	USA	19.45
Ontario	Canada	14.57
Quebec	Canada	8.49
Tamil Nadu	India	67.86

States

Ontario). The second relation is called States; it has columns named State, Country, and Pop (population of the state, in millions).

The selection of the programming language for the relational model is an interesting story. Codd could have viewed the relational model as a fundamental abstraction, like trees or graphs, that was embedded in a general-purpose language. The operations of the relational language would have been simple navigational steps, such as "find the value in a given row and column" or "given a row, find the following row." Indeed, earlier database abstractions, such as the network and hierarchical models, were exactly that. Fortunately, Codd's view was of a declarative abstraction, and that choice, which has been followed consistently as the programming language evolved, was instrumental in making the relational model the dominant approach to database management.

In the original formulation,[16] the programming language for the relational model was taken to be nonrecursive, first-order logic, or equivalently, a collection of five algebraic operations: union, set difference, selection, projection, and join, called *relational algebra*. The last three may be unfamiliar, but are defined as follows:

- Selection takes a condition C on the column names of a relation R and returns those rows of R that satisfy C. For example, if we apply the condition "Country = India" to the relation States in Figure 3, we get a new relation with the same column names State, Country, and Pop, but only the second and sixth rows of States.
- Projection takes a set of the column names for a relation and produces a new relation with the same set of rows, but only those columns.
- Join takes two relations and a condition C involving the column names of the two relations and produces a new relation by:
  1. Considering each pair of rows, one from each relation,
  2. And if the values in those two rows satisfy condition *C*, adding the concatenation of the two rows to the resulting relation.

EXAMPLE 3.1. *Suppose we join the two relations Cities and States from Figure 3 using the condition that the State columns of the two relations must agree. In principle, we must consider all 36 pairs of rows, one from each relation. However, each row of Cities will satisfy the condition only with the one row of States that has the same value of the column State. For example, the first row of Cities,*

(Toronto, Ontario)

matches only the fourth row of States, that is,

(Ontario, Canada, 14.57)

Thus, one of the six rows in the result is the concatenation of these two rows:

(Toronto, Ontario, Ontario, Canada, 14.57)

In the common case where the condition requires only equality between two attributes, we generally eliminate one of the two identical columns and would write the schema of the resulting relation as (City, State, Country, Pop) and the row just discussed as (Toronto, Ontario, Canada, 14.57).[d]

3.2. The SQL abstraction. Shortly after the relational model was proposed, a big step forward was the development of a richer programming language, called SQL.[14] In its original formulation, SQL was still not a Turing-complete language. It did, however, support more than the original relational model. The underlying data model allowed both sets and bags—that is, the same row could appear several times—and it was also possible to sort the rows of a relation based on the values in one or more columns. In addition to the relational algebra operators described previously, SQL supported *grouping and aggregation*. That is, SQL allows the programmer to group the rows of a relation by the values in one or more attributes, and then to aggregate—for instance, sum or average—the values of one or more columns in each group. Recall that Example 1.4 discussed grouping with the sum as the aggregation.

Figure 4. Grouping by Country and summing Pop.

Country	SUM (Pop)
USA	58.96
Canada	23.06
India	182.06

EXAMPLE 3.2. *Consider the relation States from Figure 3. We can group the rows by the value of the Country column and then sum, for each country, the populations of the states for that country. The resulting table is shown in Figure 4.*

As SQL developed, many more capabilities were added to the standard, including the ability to write recursive programs and the ability to invoke code in a general-purpose programming language. As a result, SQL is now, in principle, Turing-complete. Fortunately, the vast majority of SQL programs do not use the features that make it Turing-complete, Thus, in practice, it is still possible to compile SQL in a way that exploits many opportunities for optimization that we would expect for a declarative abstraction.

3.3. Compilation of SQL. A program written in SQL is normally compiled into a lower-level language, such as C. The C code makes heavy use of library functions, for example, to execute operations such as selection or join. The early phases of compilation—lexical analysis, parsing, and so on—are similar to those for any common language. The place where SQL differs from the norm is in the code optimization phase (normally called *query optimization*). Recall that optimization of a language such as C has to content itself with saving a machine instruction here and there, so a speedup by a factor of two is a good

d Now we can see how the disk-based hashing technique of Example 1.4 can be used to implement an efficient join algorithm based on hashing. We could hash all the rows of the two relations on their State components, and be assured that any two rows that joined would wind up in the same bucket. That allows us to generate the rows of the join by going bucket-by-bucket, as we did in that example.

outcome.[e] But the operations of SQL and the relational model in general are vastly more powerful than machine instructions. For instance, one operator of the syntax tree could join two huge relations as if it were one step.

As a result, in comparison with C or its ilk, a SQL program consists of relatively few steps, but those steps can each take an enormous amount of time if implemented as is. Thus, the compiler for SQL usually does an almost exhaustive search for equivalent syntax trees that will take orders of magnitude less time to execute. Even spending time that is exponential in the size of the SQL program to optimize a program that will only be executed once makes sense, because this program will typically be executed on relations that are sufficiently large to justify the expenditure of compilation time. A great deal is known about techniques for optimizing SQL or similar languages; see Garcia-Molina et al.[21] for example.

EXAMPLE 3.3. *Here is a very simple example of where query optimization can make a vast difference in the running time of a SQL program. Suppose we are given the relations Cities and States as in Figure 3, but these relations are much larger; they have rows for every city and every state on the planet. We want to find the country in which Toronto is located. Here is the SQL query one would naturally write:*

```
SELECT Country
FROM Cities, States
WHERE Cities.State = States.State
      AND City = 'Toronto';
```

The literal meaning of this program is as follows:

1. *Join the relations Cities and States, using the condition that the State values from both relations are the same—that is, perform the join described in Example 3.1.*
2. *Select from the result of (1) the row for Toronto.*
3. *Print the value of Country for the row from (2).*

This plan can be translated directly into C code and executed. But it is a disaster. It requires us to join the two relations in their entirety—a task that even if done efficiently, such as using the hash-join technique discussed in Section 1.7.1, will require running time proportional to the sum of the sizes of the relations.

However, it is possible in at least some situations to use a plan that takes constant time, independent of the size of the relations. In essence, we reverse the order of the first two steps. That is, we first use the selection operator to select from the Cities relation only the row for Toronto. Then, we take the value of State for Toronto and select from the States relation only the row for that State. Finally, we print the Country value for that row.

But we need to be a bit careful. Just because we want only the Toronto row from Cities doesn't mean we will always find it first, as was suggested in Figure 3. Rather, since relations are sets or bags, we could find that row anywhere. However, we can also use another abstraction to organize a relation to make certain operations very fast. There is a substantial theory of how to do this organization for a variety of operations, but in this case, at least one organization is obvious. We can hash the rows of Cities according to their City value. Then, given any city name, we can go immediately to the bucket that contains the row(s) for that city, without having to scan the entire relation. Likewise, we can organize the States relation into a hash table with a hash function that depends only on the State value, and make finding the row for a given state very fast.

3.4. Distributed computing abstractions. For many years, it has been recognized that the capabilities of a single processor were reaching a limit. To handle increasingly large datasets, it was necessary to develop algorithms that use many independent machines together. A number of abstractions that let us think about distributed computing algorithms have been implemented and are in serious use. In general, these abstractions share some features:

1. Their model of data is that of a conventional programming language, but the data itself is distributed among many different tasks, which we shall call *compute nodes*. In practice, several tasks may be executed at the same processor, but it is often convenient to think of the tasks as if they were processors themselves.
2. Programs are also written in a conventional language, but the same program can run simultaneously at each node.
3. There is a facility for nodes to communicate with other nodes. This communication occurs in phases and alternates with computation phases.

This class of abstractions has several different performance metrics that are of interest. The obvious one is the wall-clock time taken to execute the program(s) involved at all the nodes in parallel. But sometimes, the bottleneck is the time needed to communicate among the nodes, especially when a large amount of data needs to be shared among the nodes. A third running time issue is the number of *rounds* (one computation phase followed by one communication phase) taken by an algorithm.

3.5. The bulk-synchronous abstraction. One popular abstraction, which we shall not discuss in detail, is the *bulk-synchronous* model of Valiant.[44] This model has recently been popularized in the context of computing clusters by Google's Pregel system[32] and it has received a number of similar implementations since then. See a survey.[47]

In the bulk-synchronous model, the compute nodes can be thought of as nodes of a complete graph. In an initialization phase, each node executes an initialization program on its local data and thereby generates some messages for other specific nodes. When all computations are finished, the messages are all communicated to their destination. In the second round, all the nodes execute a "main" program on their incoming messages and their local data, which may cause additional messages to be generated. After computation subsides, these messages are communicated to their destinations, and a third round begins, where the main program is again executed on the new incoming messages. This alternation of computation and message passing continues, until at some round, no more messages are generated.

e Of course, making all the world's software run twice as fast is wonderful; we do not wish to deprecate the importance of code optimization for conventional languages.

3.6. The MapReduce abstraction. MapReduce[17] is an abstraction that has proven to be a very powerful tool for creating parallel programs without the need for the programmer to think of parallelism explicitly. The original implementation by Dean and Ghemawat at Google was made popular by Hadoop[12] and more recently by Spark.[51] Moreover, this model is able to easily support the relational model operations that typically take the most time: join and grouping/aggregation, along with many other important operations on large-scale data.

The data model for MapReduce is sets of *key-value pairs*. However, "keys" in this sense are not normally unique; they are simply the first components of the pairs. Programs in MapReduce are written in some conventional programming language, and each MapReduce *job* has two associated programs called, unsurprisingly, "Map" and "Reduce." The input to a job is a set of key-value pairs. The Map program is written to apply to a single key-value pair and it produces any number of key-value pairs as its output. The data type of the output pairs is often different from the types of the input pairs. Since Map applies to each key-value pair independently, we can create many tasks, called *mappers*, each of which takes a subset of the input pairs and applies Map to each. The mappers can thus be executed in parallel, using as many processors as we have available.

After the mappers do their work, a communication phase takes all the outputs of Map applied to all the input pairs and sorts them by key. That is, the entire collection of output key-value pairs is organized into *reducers*, each of which is a single key, say x, and a list of all the associated values, that is, the list of y's such that there is an output pair (x, y). We then execute the program Reduce on each reducer. As each reducer is independent of the others, we can organize reducers into tasks and run each task on a different processor. The output of the entire job is the set of key-value pairs produced from each of the reducers.

Example 3.4. *Let us see how we could join the two relations of Figure 3 with the condition that the State values from the two relations be the same. The MapReduce implementation of the join is essentially the same as the hash-join algorithm discussed in Example 3.1. However, instead of throwing rows into buckets, we throw them into reducers, each of which is guaranteed to have only one value of State, while buckets could contain more than one state.*

The input consists of key-value pairs (x, y), *where y is a row of the relation x. The Map function takes an input* (x, y) *and produces a single key-value pair* $(s, (x, z))$ *where s is the State component of row y, and z is the other components of row y. That is, if x is Cities, then z is just the City component of row y, but if x is States, then z is the Country and Pop components of y.*

Now, the sort-by-key operation groups all these key-value pairs—one pair for each row of either relation—into reducers. Each reducer consists of a single State value and a list of all the rows of either relation with that State. Using the data of Figure 3 there will be, associated with any State, exactly one row of States, but typically many rows of Cities. The job of the reducer for State s is to combine each row of Cities that has State = s with each row of States that has State = s. The output is a set of key-value pairs, where the key is always the name of the output relation (which is chosen by the Reduce program) and each pair has as its value one of the rows of the join.

3.7. Performance metrics for MapReduce. The notion of what makes a MapReduce algorithm good is surprisingly tricky. In an example such as Example 3.4, where the Map function produces only one key-value pair, communication is unlikely to be a bottleneck, and therefore, we can consider only the time it takes to execute all the mappers and all the reducers. The mappers and reducers can be divided among available processors, so the wall-clock time is inversely proportional to the number of processors—that is, as little time as possible, given the available hardware. But there are other applications where each mapper produces many pairs, and then, the communication can become a bottleneck, or at least an issue that needs to be taken into account. Sending a packet from one processor to another requires substantial time to form the packet and possibly a delay going through the network.

For many problems, there is a relationship between the number of key-value pairs sent to each reducer (*reducer size*) and the number of pairs produced by one mapper (*replication rate*). Typically, there is a lower bound on the product of these two quantities, but subject to that constraint there are many different algorithms to choose from. If we turn the cost of communicating a key-value pair into an amount of time, then the design theory of Afrati et al.[3] lets us pick the optimum reducer size and replication rate, subject to whatever lower-bound constraint applies to the problem at hand. Thus, we can minimize the wall-clock time due to both computation by mappers and reducers and the communication between mappers and reducers.

There have also been attempts to examine the number of *rounds* of MapReduce (one mapping phase followed by one reduce phase) needed for certain problems. This question has a pitfall, in that without restricting the way MapReduce algorithms can operate, the answer is always "one round." That is, given any serial algorithm A we can use the following:

1. A Map function that takes its input and produces a pair with key = 1 and its input pair as a value.
2. A Reduce function that applies algorithm A to the list of values associated with the key 1.

Thus, all the work is done at the single reducer associated with the key 1, and we have simply disguised a serial algorithm as a MapReduce job.

To disallow this behavior, Koutris and Suciu[29] uses a model *MPC*, which views MapReduce algorithms as characterized by the parameters R, the number of rounds, and L, the maximum amount of data any task is allowed to use. Both parameters are functions of n, the size of the input for the given problem instance. They further require that L is $O(n^{1-\epsilon})$ for some $\epsilon > 0$; that is, as the input gets large, no task can access more than a vanishing fraction of the data. Under this restriction, they are able to get lower bounds on the relationship between L and R for certain problems. Karloff et al.[23] add two additional requirements: Tasks must execute in polynomial time as a function of input size n, and the number of rounds must be polylogarithmic in n. There are

a number of interesting open problems in this model—for instance, can you find connected components of a graph of size n in $o(\log n)$ rounds?

4. Quantum Computing

Recently, there has been a lot of interest around the world in quantum computation and quantum programming languages. Quantum computing is particularly interesting because the models of computation underlying quantum programming languages are vastly different from the models of computation found in classical programming languages.

The story begins with quantum mechanics, a fundamental theory in physics developed in the early part of the 20th century that describes the physical properties of nature at the scale of atoms and subatomic particles. We will present the basic postulates of quantum mechanics from which all the laws of quantum mechanics can be derived. From these postulates, we can derive the abstraction of quantum circuits, one of the fundamental models of computation underlying quantum programming languages.

4.1. The postulates of quantum mechanics. Complex linear algebra and Hilbert spaces (a complex vector space with an inner product) are normally used to describe the postulates of quantum mechanics. Nielsen and Chuang[35] is a good source for learning about this subject. To begin, let us review a few basic definitions from complex linear algebra that are used in the postulates. It is helpful to view operators as matrices of complex numbers that act on vectors. The *Hermitian conjugate* of a matrix U is denoted $U^\dagger$; it is the conjugate-transpose of the matrix U—that is, take the transpose of U and then negate the complex part of each value.

The notion of a *unitary operator* is central to quantum mechanics. An operator U is unitary if $UU^\dagger = I$, where I is the identity. What this means is that the action of every unitary transform is *invertible*. Invertible means reversible—that is, we can reconstruct the input from the output. An operator U is said to be *Hermitian* if $U = U^\dagger$. A Hermitian operator is self-adjoint.

Postulate 1. The state space of an isolated physical system can be modeled by a Hilbert space. The state of the system is completely described by a unit vector in this state space.

Postulate 1 allows us to define the *qubit* as a unit vector in a two-dimensional state space. The qubit is the quantum computing analog of a bit (0 or 1) in classical computing. If the vectors $|0\rangle = (1, 0)$ and $|1\rangle = (0, 1)$ are used as an orthonormal basis for the two-dimensional Hilbert space, then an arbitrary state vector $|\psi\rangle$ in that space can be written as (α, β) or as

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where α and β are complex numbers. Since $|\psi\rangle$ is a unit vector, $|\alpha|^2 + |\beta|^2 = 1$.

The qubit $|\psi\rangle$ exhibits an intrinsic phenomenon of quantum mechanics called *superposition*. Unlike a bit in classical computing, which is always a 0 or a 1, without knowing α and β it is not possible to say that the qubit $|\psi\rangle$ is definitely in the state $|0\rangle$ or definitely in the state $|1\rangle$. All we can say is that it is in some combination of these two states.

Postulate 2. The evolution of the state of a closed quantum system from one time to another can be described by a unitary operator.

There is an equivalent way of formulating Postulate 2 using Schrödinger's equation. However, we will only consider the unitary formulation here, since it naturally leads to the quantum circuit model of computation.

Postulate 3. To get information from a closed quantum system, we can apply measurements to the system. A measurement returns an outcome with some probability. The sum of the probabilities of the possible outcomes is 1. A measurement changes the state of the quantum system.

We would not go into the details of Postulate 3 but for the purposes of our discussion here we can think of a measurement on a single qubit $|\psi\rangle$ as a Hermitian operator that returns the outcome 0 with probability $|\alpha|^2$ and the outcome 1 with probability $|\beta|^2$. Recall that since $|\psi\rangle$ is a unit vector, $|\alpha|^2 + |\beta|^2 = 1$. The measurement collapses the state vector to one of the two basis vectors of the 2D Hilbert space. We note that Heisenberg's famous uncertainty principle in quantum mechanics can be derived from the rules of complex linear algebra and postulates 1–3.

The fourth postulate shows how the dimension of the state space of a composite physical system grows when we combine physical systems.

Postulate 4. The state space of a composite physical system is the tensor product of the state spaces of the component physical systems.[f]

Postulate 4 shows that if we add a single qubit to a physical system, we double the number of dimensions of its state space. Thus, if we combine n single-qubit systems, we get a composite system with a state space of dimension 2^n by taking the tensor product of the state spaces of the n single-qubit systems. This exponential growth in the size of the state space makes it difficult to simulate the behavior of large quantum systems on classical computers.

4.2. Quantum circuits. From the four postulates of quantum mechanics, we can derive a model of computation called quantum circuits, which is the fundamental abstraction underlying quantum programming languages. Quantum circuits consist of gates and lines. They resemble Boolean circuits in classical computation but there are several important distinctions. It is helpful to view a quantum gate as an orthonormal matrix of complex numbers and its output as a vector obtained by applying the matrix to the input vector.

Single-qubit gates. A single-qubit gate has a line leading into the gate and a line leading out of the gate. The input line feeds a qubit $|\psi\rangle$ to the gate. The gate applies a unitary transform U to the incoming qubit $|\psi\rangle$ and delivers an outgoing qubit $U|\psi\rangle$ to the output line.

In classical Boolean circuits, there is only one nontrivial, single-bit logic gate, namely, the Boolean NOT-gate. In quantum circuits, any unitary transform in 2D complex Hilbert space can be a single-qubit quantum gate. Here are two important quantum single-qubit gates.

Example 4.1. *The* quantum-NOT gate, *often denoted X, maps the qubit* $\alpha|0\rangle + \beta|1\rangle$ *to the qubit* $\beta|0\rangle + \alpha|1\rangle$. *Basically,*

f As an example, if $A = (a_1, a_2, a_3, a_4)$ and $B = (b_1, b_2)$ are two vectors, then $A \otimes B$, the tensor product of A and B, is the vector $(a_1b_1, a_1b_2, a_2b_1, a_2b_2, a_3b_1, a_3b_2, a_4b_1, a_4b_2)$.

it flips the coefficients of the vector representing the qubit in the 2D Hilbert space. Note that $X|0\rangle = |1\rangle$ *and* $X|1\rangle = |0\rangle$.

The quantum-NOT gate *X* can be represented by the 2×2 matrix:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

EXAMPLE 4.2. *The* quantum-Hadamard gate, *denoted H, maps the qubit* $\alpha|0\rangle + \beta|1\rangle$ *to the qubit:*

$$\alpha\left((|0\rangle+|1\rangle)/\sqrt{2}\right)+\beta\left((|0\rangle-|1\rangle)/\sqrt{2}\right)$$

Note that HH = I, the identity operator.

The quantum-Hadamard gate *H* can be represented by the 2×2 matrix:

$$H = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}$$

There are many other useful single-qubit quantum gates.

Multiple-qubit gates. A multiple-qubit quantum gate has *n* input lines leading into the gate and *n* output lines emanating from the gate. The gate consists of a unitary operator *U* that can be represented by a $2^n \times 2^n$ matrix of complex numbers whose rows and columns are orthonormal.

EXAMPLE 4.3. *The controlled-NOT gate (CNOT for short) is a very useful two-qubit gate. It has two input lines and two output lines, one called the* control *and the other called the* target. *The action of the gate action is as follows. If the incoming qubit on the control is* $|0\rangle$, *the qubit on the target line passes through unchanged. If the incoming control qubit is* $|1\rangle$, *the target qubit is flipped. In either case, the control qubit does not change. If* $|\psi_1\ \psi_2\rangle$ *represents* $|\psi_1\rangle \otimes |\psi_2\rangle$, *the tensor product of qubits* $|\psi_1\rangle$ *and* $|\psi_2\rangle$, *then we can describe the action of the CNOT gate as follows:*

$$|00\rangle \rightarrow |00\rangle;\ |01\rangle \rightarrow |01\rangle$$
$$|10\rangle \rightarrow |11\rangle;\ |11\rangle \rightarrow |10\rangle$$

Circuits. We can now describe quantum circuits, a fundamental computational model underlying quantum computing and quantum programming languages. A quantum circuit is an acyclic graph consisting of lines, quantum gates, and measurement gates. Since a quantum circuit is acyclic, there can be no loops or feedback. Since logical OR is not a unitary operation, there can be no fan-in where lines are joined together. In addition, in quantum mechanics it is impossible to make a copy of an unknown quantum state (the *no-cloning theorem*), so fan-out is not possible either.

A *measurement gate* takes as input a line bringing in a single qubit in state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ and produces as output a probabilistic classical bit whose value is 0 with probability $|\alpha|^2$ or 1 with probability $|\beta|^2$.

We finish our quantum cirucuits discussion with an example that illustrates an unusual property of quantum computation: *entanglement*.

EXAMPLE 4.4. *Consider a quantum circuit that has two input lines x and y, shown in Figure 5. The x-line is connected to a Hadamard gate and the output of the Hadamard gate becomes the control line of a CNOT-gate. The y-line is the target line of the CNOT gate. We will call this the EPR quantum circuit, named after Einstein, Podolsky, and Rosen, who pointed out strange properties of the output states of this circuit. Here is the action of this circuit on four values of the two input qubits* $|xy\rangle$*:*

$$|00\rangle \rightarrow \frac{|00\rangle+|11\rangle}{\sqrt{2}}$$
$$|01\rangle \rightarrow \frac{|01\rangle+|10\rangle}{\sqrt{2}}$$
$$|10\rangle \rightarrow \frac{|00\rangle+|11\rangle}{\sqrt{2}}$$
$$|11\rangle \rightarrow \frac{|01\rangle+|10\rangle}{\sqrt{2}}$$

You can describe the operation of a quantum circuit as the sequence of state vectors showing the state of the quantum system after the application of each stage of gates. For Figure 5, we would have the following:

1. Before H-gate: $|00\rangle$
2. After H-gate before CNOT-gate: $\frac{|0\rangle+|1\rangle}{\sqrt{2}} \otimes |0\rangle$
3. After CNOT-gate: $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$

A state of a composite quantum system that cannot be written as a tensor product of the states of its component systems is said to be *entangled*. It can be shown that the EPR output states above are entangled. There are no two single-qubit states $|\psi_1\rangle$ and $|\psi_2\rangle$ such that the state

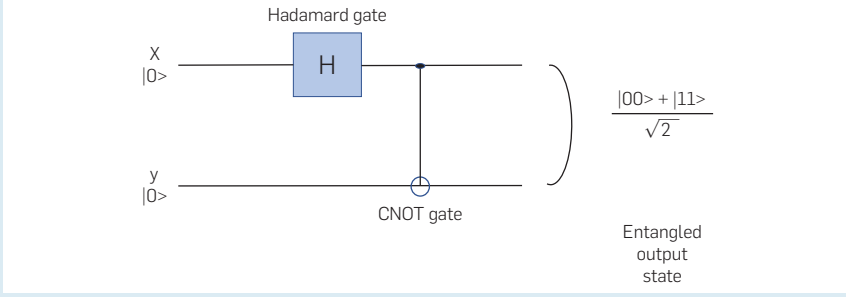
$$\frac{|00\rangle+|11\rangle}{\sqrt{2}} = |\psi_1\ \psi_2\rangle.$$

Entanglement plays a critical role in quantum computing, but the physical phenomenon of entanglement is still a mystery to physicists. In fact, Einstein called it "spooky action at a distance."

4.3. Quantum algorithms. A quantum computing device will most likely be used as an adjunct controlled by a classical computer. Programs for quantum computers are often expressed as a mixture of classical computation and quantum algorithms. A quantum algorithm is frequently presented as a quantum circuit having the following structure:

1. The circuit starts off with all the input qubits set to a particular state, usually $|0\rangle$.

Figure 5. Quantum circuit to generate EPR state from input $|00\rangle$.



2. The circuit is put into a superposition of states.
3. The circuit acts on this superposition with unitary gates.
4. Measurements are done on the qubits in the circuit by measurement gates returning classical bits (0s and 1s) as outputs to the controlling classical computer.

Quantum computing received an enormous boost in 1994, when Peter Shor at Bell Labs published an algorithm for factoring an n-bit integer with a hybrid classical computer/quantum computer using $O(n^3)$ operations.[39] Even today, no polynomial-time algorithm is known for factoring integers on a classical computer.

Shor used classical number theory to reduce the integer-factoring problem to the problem of *order finding*. The order-finding problem is as follows: given positive integers x and N, with $x < N$ and x coprime to N, find the smallest positive integer r such that $x^r \bmod N = 1$. The integer r is called the order of x in N. For example, the order of 5 in 21 is 6 because 6 is the smallest positive integer such that $5^6 \bmod 21 = 1$.

Shor devised a quantum algorithm to solve the order-finding problem with a polynomial number of quantum gates. There is no known algorithm for solving the order-finding problem on a classical computer in polynomial time.

Quantum algorithms often use specialized techniques not found in algorithms for conventional computers. Shor's algorithm, for example, uses the quantum Fourier transform as part of its order-finding computation.

5. Future Directions

As this article argues, abstractions have had considerable impact on many fields within computer science. But the story of abstractions in computer science has many more papers. Here are some directions that may prove interesting to theoreticians and also have practical importance.

5.1. The quantum future. Quantum computing is still a field in its infancy. Although quantum circuits can approximate arbitrary unitary operators to any desired degree of accuracy, today's quantum gate computers have only 50–100 usable qubits. In addition, there are only a handful of practically useful quantum algorithms, so much more work needs to be done in both the hardware and algorithm areas of quantum computing to overcome these limitations.[33]

On the theoretical side, many open questions also remain. For example, if we could prove the problem of not being able to factor integers on a classical computer in polynomial time, then we would have an example of a problem that a quantum computer can solve faster than a classical computer. This is but one of many deep theoretical problems that remain open. You might wish to consult Aaronson[1] for a list of algorithmic challenges in the quantum abstraction.

A number of full-stack quantum computing programming languages have been developed. A PhD student at Columbia University, Krysta Svore,[40] showed that the compiler architecture discussed in Section 2 could be incorporated with error correction into a layered software architecture for quantum computing design tools. Upon graduation she joined Microsoft Research, where she and her colleagues subsequently developed the Q# quantum programming language, which is part of the Microsoft Quantum Development Kit.[41] Along with Q#, Wikipedia under "Quantum programming" now lists more than a dozen quantum programming languages.

5.2. Abstractions for computer systems and hardware. The success of MapReduce and other high-level abstractions for a particular kind of computing platform (the computing cluster, in this case) suggests that there might be similar abstractions for other platforms. For example, there is current interest in *serverless computing*,[49] where the data is kept solely in a file system, and computing is done by renting one or more servers for short periods of time.

On a smaller scale, special-purpose hardware is a growing trend, and is likely to play an increasingly important role in speeding up the execution of important algorithms on large-scale data. You have probably heard of graphics processing units (GPUs) and field-programmable gate arrays (FPGAs). Plasticine[37] is another sort

of chip designed to support high communication bandwidth along with parallelism and is likely to be available soon. It would be useful to have high-level abstractions that matched these architectures, in the sense that algorithms written using these abstractions can be compiled into efficient implementations using one or more of these chip types.

5.3. Abstraction taxonomy. Over the course of many years, powerful abstractions associated with programming language processing have been invented that have helped transform the field of compiler design from an art to a science. But the final paper has yet to be written. It would be useful to extend our rudimentary taxonomy for abstractions in Section 1.2 to cover more of the field of programming languages and compilers, and indeed more of computer science. Abstractions associated with continuously running systems, such as operating systems, networks, and the Internet, would be natural ones to include.

Further, we hope that a taxonomy is useful for more than organizing lectures in a data structure course. In particular, we hope that there will be a study of what makes one abstraction more useful than another. For instance, we mentioned in Section 3.1 how the relational model naturally became a declarative abstraction, while prior database models did not lend themselves to languages such as SQL, which enabled very high-level programming. Similarly, regular expressions seem very well suited to describing programming language tokens and other interesting sets of strings, while equivalent notations, such as Chomsky's type-3 grammars (a special case of CFGs) never found much use in applications such as lexical analysis. It is natural to ask, "Why?"

One intriguing new area is the use of machine learning to create software applications using data rather than source programs written in some programming language. In a sense, machine learning is a way to create software that does not involve traditional compilation. Abstractions that could guide the efficient creation of robust applications using machine learning would be very beneficial.

Acknowledgments

We appreciate the careful reading and comments of Peter Denning. We also wish to acknowledge the comments and help from several anonymous referees.

References

1. Aaronson, S. Ten semi-grand challenges for quantum computing theory. (2005), https://www.scottaaronson.com/writings/qchallenge.html.
2. Adel'son-Vel'skii, G.M. and Landis, E.M. An algorithm for organization of information. In *Doklady Akademii Nauk 146* (1962), Russian Academy of Sciences, 263–266.
3. Afrati, F.N., Sarma, A.D., Salihoglu, S., and Ullman, J.D. Upper and lower bounds on the cost of a map-reduce computation. In *Proceedings of the VLDB Endowment 6*, 4 (2013), 277–288.
4. Aho., A.V. *Algorithms for Finding Patterns in Strings*. MIT Press, Cambridge, MA, USA (1991), 255–300.
5. Aho., A.V. Teaching the compilers course. *SIGCSE Bull 40*, 4 (Nov. 2008), 6–8.
6. Aho., A.V. and Corasick, M.J. Efficient string matching: An aid to bibliographic search. *Comm. ACM 18*, 6 (June 1975), 333–340.
7. Aho., A.V., Hopcroft, J.E., and Ullman, J.D. *Data Structures and Algorithms*. Addison-Wesley, (1983).
8. Aho., A.V., Johnson, S.C., and Ullman, J.D. Deterministic parsing of ambiguous grammars. In *Conf. Record of the ACM Symposium on Principles of Programming Languages*. P.C. Fischer and J.D. Ullman, Eds. ACM Press, (Oct. 1973), 1–21.
9. Aho, A.V., Lam, M.S., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools, 2nd Edition*. Addison-Wesley Longman Publishing Co., Inc., USA, (2006).
10. Aho, A.V. and Ullman, J.D. *Foundations of Computer Science*. W.H. Freeman and Co., USA (1994).
11. Allen, F. Control flow analysis. *Sigplan Not 5* (1970), 1–19.
12. Apache Software Foundation. Hadoop.
13. Bayer, R. and McCreight, E.M. Organization and maintenance of large ordered indices. In *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access*, 107–141.
14. Chamberlin, D.D. and Boyce, R.F. Sequel: A structured English query language. *SIGFIDET '74*, Association for Computing Machinery (1974), 249–264.
15. Codd, E.F. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June 1970), 377–387.
16. Codd, E.F. Relational completeness of database sublanguages. In *Database Systems*. Prentice-Hall (1972), 65–98.
17. Dean, J. and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004), 137–150.
18. Denning, P. and Tedre, M. *Computational Thinking. MIT Press Essential Knowledge Series*. MIT Press (2019).
19. Elgot, C.C. and Robinson, A. Random-access stored-program machines, an approach to programming languages. *J. of the ACM 11*, 4 (Oct. 1964), 365–399.
20. Eppstein, D. and Galil, Z. Parallel algorithmic techniques for combinatorial computation. *Annual Reviews Inc.* (1988), 233–283.
21. Garcia-Molina, H., Ullman, J.D., and Widom, J. *Database Systems: The Complete Book*. Prentice Hall Press, USA (2008).
22. Guibas, L.J. and Sedgewick, R. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science* (1978), 8–21.
23. Karlo, H., Suri, S., and Vassilvitskii, S. A model of computation for MapReduce (2010), 938–948.
24. Kildall, G.A. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages* (1973), 194–206.
25. Kitsuregawa, M., Tanaka, H., and Moto-Oka, T. Application of hash to database machine and its architecture. *New Gener. Comput 1*, 1 (1983), 63–74.
26. Kleene, S.C. Representation of events in nerve nets and finite automata, *Automata Studies 34*, Princeton University Press, Princeton, NJ, USA (1956), 37–40.
27. Knuth, D.E. On the translation of languages from left to right. *Inf. Control 8*, 6 (1965), 607–639.
28. Korenjak, A.J. A practical method for constructing LR(k) processors. *Commun. ACM 12*, 11 (Nov. 1969), 613–623.
29. Koutris, P. and Suciu, D. A guide to formal analysis of join processing in massively parallel systems. *ACM SIGMOD Rec 45*, 4 (2016), 18–27.
30. Lesk, M.E. and Schmidt, E. *Lex—A Lexical Analyzer Generator*. W.B. Saunders Company, USA (1990), 375–387.
31. Liskov, B. and Zilles, S. Programming with abstract data types. *SIGPLAN Not 9*, 4 (Mar. 1974), 50–59.
32. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G. Pregel: A system for large-scale graph processing. *SIGMOD '10*, Association for Computing Machinery (2010), 135–146.
33. Martonosi M. and Roetteler, M. Next steps in quantum computing: Computer science's role. Computing Community Consortium (2019).
34. McNaughton, R. and Yamada, H. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers 9*, 1 (1960), 39–47.
35. Nielsen, M.A. and Chuang, I.L. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, USA (2011).
36. Peterson, W.W. Addressing for random-access storage. *IBM J. of Res. Dev 1*, 2 (1957), 130–146.
37. Prabhakar, R., Zhang, Y., Koeplinger, D., Feldman, M., Zhao, T., Hadjis, S., Pedram, A., Kozyrakis, C., and Olukotun, K. Plasticine: A reconfigurable architecture for parallel patterns. In *Proceedings of the 44th Annual Intern. Symp. on Computer Architecture* (2017), 389–402.
38. Rabin, M.O. and Scott, D. Finite automata and their decision problems. *IBM J. of Res. Dev, 3* (Apr. 1959), 114–125.
39. Shor, P.W. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual Symp. on Foundations of Computer Science* (1994), 124–134.
40. Svore, K.M., Aho, A.V., Cross, A.W., Chuang, I.L., and Markov, I.L. A layered software architecture for quantum computing design tools. *Computer 39*, 1 (2006), 74–83.
41. Svore, K.M., Geller, A., Troyer, M., Azariah, J., Granade, C.E., Heim, B., Kliuchnikov, V., Mykhailova, M., Paz, A., and Roetteler, M. Q#: Enabling scalable quantum computing and development with a high-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop* (Feb. 2018), 7:1–7:10.
42. Thompson, K. Programming techniques: Regular expression search algorithm. *Commun. ACM 11*, 6 (June 1968), 419–422.
43. Turing, A.M. On computable numbers, with an application to the Entscheidungsproblem. In *Proc. London Math. Soc 2*, 42 (1936), 230–265.
44. Valiant, L.G. A bridging model for parallel computation. *Commun. ACM 33*, 8 (Aug. 1990), 103–111.
45. Vyssotsky, V. and Wegner, P. A graph theoretical Fortran source language analyzer. Bell Laboratories Internal Document (1963).
46. *Wikipedia*, 2021. https://en.wikipedia.org/wiki/Abstract_data_type.
47. *Wikipedia*, 2021. https://en.wikipedia.org/wiki/Bulk_synchronous_parallel.
48. *Wikipedia*, 2021. https://en.wikipedia.org/wiki/Interface_(Java).
49. *Wikipedia*, 2021. https://en.wikipedia.org/wiki/Serverless_computing.
50. Wing, J.M. Computational thinking. *Commun. ACM 49*, 3 (2006), 33–35.
51. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. Apache spark: A unified engine for big data processing. *Commun. ACM 59*, 11 (Oct. 2016), 56–65.

Alfred Aho is Lawrence Gussman Professor Emeritus of Computer Science at Columbia University, New York, NY.

Jeffrey Ullman is computer science professor emeritus at Stanford University in Stanford, CA.

Watch the authors discuss this work in the exclusive *Communications* video. https://cacm.acm.org/videos/abstractions-algorithms-compilers