Unit 3: Computation

Computation: If it involves a computer, a program running on a computer and numbers going in and out then **computation** is likely happening.

The notion that mental states and processes intervene between stimuli and responses sometimes takes the form of a "computational" metaphor or analogy, which is often used as the identifying mark of contemporary cognitive science: The mind is to the brain as software is to hardware; mental states and processes are (like) computer programs implemented (in the case of humans) in brain states and processes. Some cognitive scientists make the stronger claim that mental states and processes *are* (expressible as) algorithms: "cognition *is* a type of computation" (Pylyshyn 1985: xiii). Others make a weaker, but more general, claim that cognition is comput*able*, i.e., that there are algorithms that have the same input–output behavior as cognitive processes (cf. Rapaport, forthcoming).

Thus, according to the computational view of cognitive science, (1) there are mental states and processes intervening between input stimuli and output responses, (2) these mental states and processes either *are* computations or else are comput*able*, and—hence—(3) in contrast to behaviorism, mental states and processes are capable of being investigated scientifically (even if they are not capable of being directly observed).

Insofar as the methods of investigation are taken to be computational in nature, computer science in general and artificial intelligence in particular have come to play a central role in cognitive science. It is, however, a role not without controversial philosophical implications: For if mental states and processes can be expressed as algorithms, then they are capable of being implemented in non-human computers. The philosophical issue is simply this: Are computers executing such algorithms merely simulating mental states and processes, or are they actually exhibiting them? Do such computers think?

In computability theory and computational complexity theory, a **model of computation** is the definition of the set of allowable operations used in computation and their respective costs. Only assuming a certain model of computation is it possible to analyze the computational resources required, such as the execution time or memory space or to discuss the limitations of algorithms or computers.

Some examples of models include Turing machines, recursive functions, lambda calculus, and production systems.

Models of computation:

Artificial neural networks can be considered as just another approach to the problem of computation. The first formal definitions of computability were proposed in the 1930s and '40s and at least five different alternatives were studied at the time. The computer era was started, not with one single approach, but with a contest of alternative computing models. We all know that the von Neumann computer emerged as the undisputed winner in this confrontation, but its triumph did not lead to the dismissal of the other computing models. Figure 1.1 shows the five principal contenders:

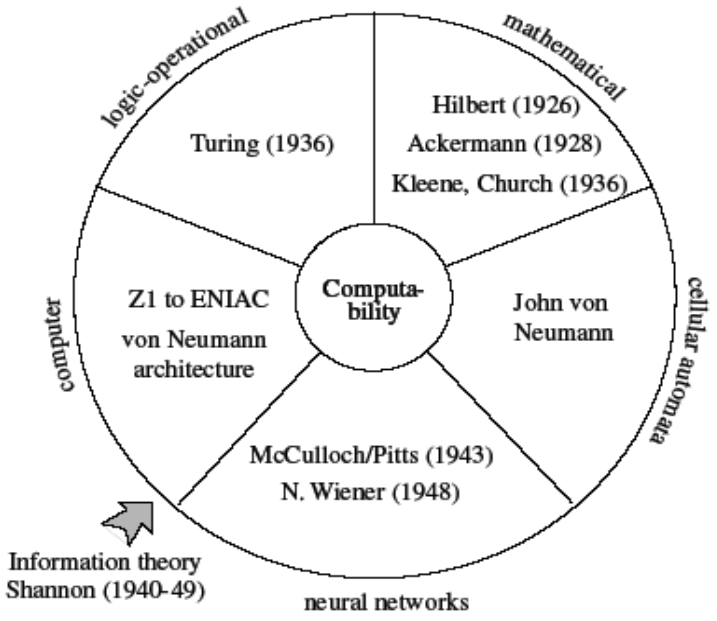


Fig. 1.1. Five models of computation

- The Mathematical Model:

David Hilbert, the famous German mathematician, was the first to state the **conjecture** that a **certain class of functions contains all intuitively computable functions**. Hilbert was referring to the primitive recursive functions, the class of functions which can be constructed from the zero and successor function using composition, projection, and a deterministic number of iterations (primitive recursion). However, in 1928, **Wilhelm Ackermann** was able to find **a computable function** which is not primitive recursive. This led to the definition of the general recursive functions. In this formalism, a new composition rule has to be introduced, the so-called μ operator, which is equivalent to an indeterminate recursion or a lookup in an infinite table. At the same time **Alonzo Church** and collaborators developed the **lambda calculus**, another alternative to the mathematical definition of the computability concept. In 1936, **Church and Kleene** were able to show that the general recursive functions can be expressed in the formalism of the lambda calculus. This led to the Church thesis that computable functions are the general recursive functions.

- The Logic-Operational Model (Turing Machine):

Alan Turing introduced another kind of computing model. The advantage of his approach is that it consists in an operational, mechanical model of computability. A Turing machine is composed of an **infinite tape**, in which symbols can be stored and read again. A read-write head can move to the left or to the right according to its internal state, which is updated at each step. The Turing thesis states that computable functions are those which can be computed with this kind of device. It was formulated concurrently with the Church thesis and Turing was able to show almost immediately that they are equivalent. The Turing approach

made clear for the first time what "programming" means, curiously enough at a time when no computer had yet been built.

- The Computer Model:

The first electronic computing devices were developed in the 1930s and '40s. Since then, "**computation-with-the-computer**" has been regarded as computability itself. Computers of the time, like the **Mark I** built at Harvard, could iterate a constant number of times but were incapable of executing open-ended iterations (WHILE loops). Therefore the Mark I could compute the primitive but not the general recursive functions. Also the **ENIAC**, which is usually hailed as the world's first electronic computer, was incapable of dealing with open-ended loops, since iterations were determined by specific connections between modules of the machine. It seems that the first universal computer was the Mark I built in Manchester. This machine was able to cover all computable functions by making use of conditional branching and self-modifying programs.

- Cellular Automata:

The history of the development of the first mechanical and electronic computing devices shows how difficult it was to reach a consensus on the architecture of universal computers. Aspects such as the economy or the dependability of the building blocks played a role in the discussion, but the main problem was the definition of the minimal architecture needed for universality. **In machines like the Mark I and the ENIAC there was no clear separation between memory and processor, and both functional elements were intertwined. Some machines still worked with base 10 and not 2, some were sequential and others parallel**. John von Neumann, who played a major role in defining the architecture of sequential machines, analyzed at that time a new computational model which he called **cellular automata**. **Such automata operate in a "computing space" in which all data can be processed simultaneously. The main problem for cellular automata is communication and coordination between all the computing** cells. This can be guaranteed through certain algorithms and conventions. It is not difficult to show that all computable functions, in the sense of Turing, can also be computed with cellular automata, even of the one-dimensional type, possessing only a few states. Turing himself considered this kind of computing model at one point in his career.

- The biological model (neural networks):

The explanation of important aspects of the physiology of neurons set the stage for the formulation of **artificial neural network models which do not operate sequentially, as Turing machines do**. Neural networks have a hierarchical multilayered structure which sets them apart from cellular automata, so that information is transmitted not only to the immediate neighbors but also to more distant units. In artificial neural networks one can connect each unit to any other. In contrast to conventional computers, no program is handed over to the hardware – such a program has to be created, that is, the free parameters of the network have to be found adaptively.

Elements of a computing model:

What are the elementary components of any conceivable computing model? In the theory of general recursive functions, for example, it is possible to reduce any computable function to some composition rules and a small set of primitive functions. For a universal computer, we ask about the existence of a minimal and sufficient instruction set. For an arbitrary computing model the following metaphoric expression has been proposed:

Computation = storage + transmission + processing.

The mechanical computation of a function presupposes that these three elements are present, that is, that data can be stored, communicated to the functional units of the model and transformed. It is implicitly assumed that a certain coding of the data has been agreed upon.

Modern computers transform storage of information into a form of information transmission. Static memory chips store a bit as a circulating current until the bit is read. Turing machines store information in an infinite tape, whereas transmission is performed by the read-write head. Cellular automata store information in each cell, which at the same time is a small processor.

Self Study: DFA, NFA, and PDA (with practical examples)

The Turing Machine:

A **Turing machine** is a theoretical device that manipulates symbols contained on a strip of tape. Despite its simplicity, a Turing machine can be adapted to simulate the logic of any computer algorithm, and is particularly useful in explaining the functions of a CPU inside of a computer. The "Turing" machine was described by **Alan Turing** in 1937, who called it an "automatic-machine". Turing machines are not intended as a practical computing technology, but rather as a **thought experiment** representing a computing machine. They help computer scientists understand the limits of mechanical computation.

The Turing machine mathematically models a machine that mechanically operates on a tape on which symbols are written which it can read and write one at a time using a tape head. Operation is fully determined by a finite set of elementary instructions such as "in state 2, if the symbol seen is 0, write a 1; if the symbol seen is 1, shift to the right, and change into state 3; in state 3, if the symbol seen is 0, write a 1 and change to state 4;" etc.

More precisely, a Turing machine consists of:

1. A **TAPE** which is divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. The alphabet contains a special *blank* symbol (here written as '0') and one or more other symbols. The tape is assumed to be arbitrarily extendable to the left and to the right, i.e., the Turing machine is always supplied with as much tape as it needs for its computation. Cells that have not been written to before are assumed to be filled with the blank symbol. In some models the tape has a

left end marked with a special symbol; the tape extends or is indefinitely extensible to the right.

2. A **HEAD** that can read and write symbols on the tape and move the tape left and right one (and only one) cell at a time. In some models the head moves and the tape is stationary.

3. A finite **TABLE** ("action table", or *transition function*) of instructions : $q_i a_j \rightarrow q_{i1} a_{j1} d_k$, that, given the *state*(q_i) the machine is currently in *and* the *symbol*(a_j) it is reading on the tape (symbol currently under HEAD) tells the machine to do the following in sequence:
    - Either erase or write a symbol (instead of a_j written a_{j1}), *and then*
    - Move the head (which is described by d_k and can have values: 'L' for one step left *or* 'R' for one step right *or* 'S' for staying in the same place), *and then*
    - Assume the same or a *new state* as prescribed (go to state q_{i1}).

4. A **STATE REGISTER** that stores the state of the Turing table, one of finitely many. There is one special *start state* with which the state register is initialized. These states, writes Turing, replace the "state of mind" a person performing computations would ordinarily be in.

Note that every part of the machine—its state and symbol-collections—and its actions—printing, erasing and tape motion—is finite, discrete and distinguishable; it is the potentially unlimited amount of tape that gives it an unbounded amount of storage space.

Formal Definition:

Formally, we can define a Turing machine as a 7-tuple $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$ where

- Q is a finite set of *states*
- Γ is a finite set of the *tape alphabet/symbols*
- $b \in \Gamma$ is the *blank symbol* (the only symbol allowed to occur on the tape infinitely often at any step during the computation)
- $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of *input symbols*
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is a function called the ***transition function***, where L is left shift, R is right shift. (A relatively uncommon variant allows "no shift", say N / S as stationary, as a third element of the latter set.)
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is the set of *final* or *accepting states*.

Anything that operates according to these specifications is a Turing machine.

Illustration of TM can be done either through **transition table** or **transition diagram**.

Comparison of TM with real machines:

It is often said that Turing machines, unlike simpler automata, are as powerful as real machines, and are able to execute any operation that a real program can. What is missed in this statement is that, because a real machine can only be in finitely many *configurations*, in fact this "real machine" is nothing but a **deterministic finite automaton**. On the other hand, Turing machines are equivalent to machines that have an unlimited amount of storage space for their computations. In fact, Turing machines are not intended to model computers, but rather they are intended to model computation itself; historically, computers, which compute only on their (fixed) internal storage, were developed only later.

There are a number of ways to explain why Turing machines are useful models of real computers:

1. Anything a real computer can compute, a Turing machine can also compute. For example: "A Turing machine can simulate any type of subroutine found in programming languages, including recursive procedures and any of the known parameter-passing mechanisms". A large enough FSA can also model any real computer, disregarding IO. Thus, a statement about the limitations of Turing machines will also apply to real computers.

2. The difference lies only with the ability of a Turing machine to manipulate an unbounded amount of data. However, given a finite amount of time, a Turing machine (like a real machine) can only manipulate a finite amount of data.

3. Like a Turing machine, a real machine can have its storage space enlarged as needed, by acquiring more disks or other storage media. If the supply of these runs short, the Turing machine may become less useful as a model. But the fact is that neither Turing machines nor real machines need astronomical amounts of storage space in order to perform useful computation. **The processing time required is usually much more of a problem.**

4. Descriptions of real machine programs using simpler abstract models are often much more complex than descriptions using Turing machines. For example, a Turing machine describing an algorithm may have a few hundred states, while the equivalent **deterministic finite automaton** on a given real machine has quadrillions. This makes the DFA representation infeasible to analyze.

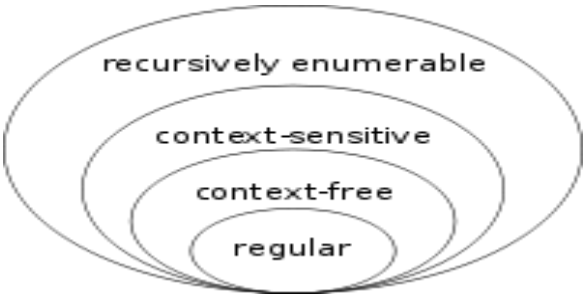
5. Turing machines describe algorithms independent of how much memory they use. There is a limit to the memory possessed by any current machine, but this limit can rise arbitrarily in time. Turing machines allow us to make statements about algorithms which will (theoretically) hold forever, regardless of advances in *conventional* computing machine architecture.

6. Turing machines simplify the statement of algorithms. Algorithms running on Turing-equivalent abstract machines are usually more general than their counterparts running on real machines, because they have arbitrary-precision data types available and never have

to deal with unexpected conditions (including, but not limited to, running out of memory).

Chomsky Hierarchy:

When **Noam Chomsky** first formalized generative grammars in 1956, he classified them into types now known as the Chomsky hierarchy. The difference between these types is that they have increasingly strict production rules and can express fewer formal languages. The Chomsky hierarchy consists of the following levels:



- **Type-0 grammars (unrestricted grammars)** include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine. These languages are also known as the recursively enumerable languages. Note that this is different from the recursive languages which can be *decided* by an always-halting Turing machine.

- **Type-1 grammars (context-sensitive grammars)** generate the context-sensitive languages. These grammars have rules of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$ with A a nonterminal and α, β and γ strings of terminals and nonterminals. The strings α and β may be empty, but γ must be nonempty. The rule $S \rightarrow \epsilon$ is allowed if S does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automaton (a nondeterministic Turing machine whose tape is bounded by a constant times the length of the input.)

- **Type-2 grammars (context-free grammars)** generate the context-free languages. These are defined by rules of the form $A \rightarrow \gamma$ with A a nonterminal and γ a string of terminals and nonterminals. These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton. Context-free languages are the theoretical basis for the syntax of most programming languages.

- **Type-3 grammars (regular grammars)** generate the regular languages. Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed (or preceded, but not both in the same grammar) by a single nonterminal. The rule $S \rightarrow \epsilon$ is also allowed here if S does not appear on the right side of any rule. These languages are exactly all languages that can be decided by a finite state automaton. Additionally, this family of

formal languages can be obtained by regular expressions. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

Unrestricted grammar (Type 0):

In formal language theory, an **unrestricted grammar** is a formal grammar on which no restrictions are made on the left and right sides of the grammar's productions.

An **unrestricted grammar** is a formal grammar $G = (N, \Sigma, P, S)$, where N is a set of nonterminal symbols, Σ is a set of terminal symbols, N and Σ are disjoint (actually, this is not strictly necessary, because unrestricted grammars make no real distinction between nonterminal and terminal symbols, the designation exists purely so that one knows when to stop when trying to generate sentential forms of the grammar), P is a set of production rules of the form $\alpha \rightarrow \beta$ where α and β are strings of symbols in $N \cup \Sigma$ and α is not the empty string, and $S \in N$ is a specially designated start symbol. As the name implies, there are no real restrictions on the types of production rules that unrestricted grammars can have.

Example. The following grammar is unrestricted. $S \rightarrow TbC$
$Tb \rightarrow \mathfrak{c}$
$cC \rightarrow Sc \mid \Lambda.$
This grammar is not context-sensitive, not context-free, and not regular.

In unrestricted grammar, the left hand side of a rule contains a string of terminals and non-terminals (at least one of which must be a non-terminal).

Unrestricted grammars and Turing machines

It may be shown that unrestricted grammars characterize the recursively enumerable languages. This is the same as saying that for every unrestricted grammar *G* there exists some Turing machine capable of recognizing *L*(*G*) and vice-versa. Given an unrestricted grammar, such a Turing machine is simple enough to construct, as a two-tape nondeterministic Turing machine. The first tape contains the input word *w* to be tested, and the second tape is used by the machine to generate sentential forms from *G*. The Turing machine then does the following:

1. Start at the left of the second tape and repeatedly choose to move right or select the current position on the tape.
2. Non deterministically choose a production $\beta \rightarrow \gamma$ from the productions in *G*.
3. If β appears at some position on the second tape, replace β by γ at that point, possibly shifting the symbols on the tape left or right depending on the relative lengths of β and γ (e.g. if β is longer than γ, shift the tape symbols left).
4. Compare the resulting sentential form on tape 2 to the word on tape 1. If they match, then the Turing machine accepts the word. If they don't go back to step 1.

It is easy to see that this Turing machine will generate all and only the sentential forms of G on its second tape after the last step is executed an arbitrary number of times, thus the language $L(G)$ must be recursively enumerable.

The reverse construction is also possible. Given some Turing machine, it is possible to create an unrestricted grammar.

Context Sensitive Grammar (Type 1):

A **context-sensitive grammar** is a formal grammar in which the left-hand sides and right-hand sides of any production rules may be surrounded by a context of terminal and nonterminal symbols. Context-sensitive grammars are more general than context-free grammars but still orderly enough to be parsed by a linear bounded automaton.

A formal grammar $G = (N, \Sigma, P, S)$ is context-sensitive if all rules in P are of the form

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

where $A \in N$ (i.e., A is a single non terminal), $\alpha, \beta \in (N \cup \Sigma)^*$ (i.e., α and β are strings of nonterminals and terminals) and $\gamma \in (N \cup \Sigma)^+$ (i.e., γ is a nonempty string of nonterminals and terminals).

Some definitions also add that for any production rule of the form $u \rightarrow v$ of a context-sensitive grammar, it shall be true that $|u| \leq |v|$. Here $|u|$ and $|v|$ denote the length of the strings respectively.

In addition, a rule of the form

$$S \rightarrow \lambda$$ provided S does not appear on the right side of any rule

where λ represents the empty string is permitted. The addition of the empty string allows the statement that the context sensitive languages are a proper superset of the context free languages, rather than having to make the weaker statement that all context free grammars with no $\rightarrow\lambda$ productions are also context sensitive grammars.

The name *context-sensitive* is explained by the α and β that form the context of A and determine whether A can be replaced with γ or not. This is different from a context-free grammar where the context of a nonterminal is not taken into consideration.

If the possibility of adding the empty string to a language is added to the strings recognized by the noncontracting grammars (which can never include the empty string) then the languages in these two definitions are identical.

Example. The following grammar is context-sensitive.

$S \to aTb \mid ab$
$aT \to aaTb \mid ac.$

Quiz. What is the language of the grammar?

Answer: $\{ab\} \cup \{a^{n+1}cb^{n+1} \mid n \in \mathbf{N}\}$. This language is context-free. For example, it has the grammar $S \to aTb \mid ab$, and $T \to aTb \mid c$.

Any context-free language is context sensitive.

Example. $\{a^nb^nc^n \mid n \geq 0\}$ is context-sensitive but not context-free. Here is a csg.

$S \to \Lambda \mid abc \mid aTBc$
$T \to abC \mid aTBC$
$CB \to CX \to BX \to BC$
$bB \to bb.$
$Cc \to cc.$

Quiz. Derive *aaabbbccc*.

Answer: $S \Rightarrow aTBc \Rightarrow aaTBCBc \Rightarrow aaabCBCBc \Rightarrow aaabBCCBc \Rightarrow aaabBCBCc$
$\Rightarrow aaabBBCCc \Rightarrow aaabbBCCc \Rightarrow aaabbbCCc \Rightarrow aaabbbCcc \Rightarrow aaabbbccc.$

Similarly the grammar for $\{a^nb^nc^n : n \geq 1\}$ can be written as:

1. $S \to aSBC$
2. $S \to aBC$
3. $CB \to HB$
4. $HB \to HC$
5. $HC \to BC$
6. $aB \to ab$
7. $bB \to bb$
8. $bC \to bc$
9. $cC \to cc$

The generation chain for aaa bbb ccc is:

S
$\Rightarrow_1 aSBC$
$\Rightarrow_1 a\boldsymbol{aSBC}BC$
$\Rightarrow_2 aa\boldsymbol{aBC}BCBC$
$\Rightarrow_3 aaaB\boldsymbol{HB}CBC$
$\Rightarrow_4 aaaB\boldsymbol{HC}CBC$
$\Rightarrow_5 aaaB\boldsymbol{BC}CBC$
$\Rightarrow_3 aaaBBC\boldsymbol{HB}C$
$\Rightarrow_4 aaaBBC\boldsymbol{HC}C$
$\Rightarrow_5 aaaBBC\boldsymbol{BC}C$
$\Rightarrow_3 aaaBB\boldsymbol{HB}CC$
$\Rightarrow_4 aaaBB\boldsymbol{HC}CC$

$$\Rightarrow_5 aaaBB\boldsymbol{BC}CC$$
$$\Rightarrow_6 aa\boldsymbol{ab}BBCCC$$
$$\Rightarrow_7 aaa\boldsymbol{bb}BCCC$$
$$\Rightarrow_7 aaab\boldsymbol{bb}CCC$$
$$\Rightarrow_8 aaabb\boldsymbol{bc}CC$$
$$\Rightarrow_9 aaabbb\boldsymbol{cc}C$$
$$\Rightarrow_9 aaabbbc\boldsymbol{cc}$$

Context Free Grammar (Type 2):

In formal language theory, a **context-free grammar** (**CFG**), sometimes also called a **phrase structure grammar** is a grammar which naturally generates a formal language in which clauses can be nested inside clauses arbitrarily deeply, but where grammatical structures are not allowed to overlap.

The canonical example is matching parentheses: parentheses of different types must open and close correctly inside each other, like this:

([[[ ]() [ ][ ]]]([ ])])

like any context free grammars, the logical units, the contents of corresponding matched parentheses, nest cleanly.

In terms of production rules, every production of a context free grammar is of the form

$V \rightarrow w$

where *V* is a single nonterminal symbol, and *w* is a string of terminals and/or nonterminals (*w* can be empty).

Context-free grammars play a central role in the description and design of programming languages and compilers. They are also used for analyzing the syntax of natural languages. Noam Chomsky has posited that all human languages are based on context free grammars at their core, with additional processes that can manipulate the output of the context free component.

Formal Definition:

A context-free grammar *G* is defined by 4-tuple: $G = (V, \Sigma, R, S)$ where

- V is a finite set of *non-terminal* characters or variables. They represent different types of phrase or clause in the sentence. They are sometimes called syntactic categories. Each variable represents a language.

- Σ is a finite set of *terminal*s, disjoint from V, which make up the actual content of the sentence. The set of terminals is the alphabet of the language defined by the grammar.

- R is a relation from V to $(V \cup \Sigma)^*$ such that $\exists w \in (V \cup \Sigma)^* : (S, w) \in R$
These relations called productions or rewrite rules.

- S is the start variable (or start symbol), used to represent the whole sentence (or program). It must be an element of V.

The language of a grammar is $G = (V, \Sigma, R, S)$ the set

$$L(G) = \{w \in \Sigma^* : S \overset{*}{\Rightarrow} w\}$$

A language L is said to be a context-free language (CFL) if there exists a CFG, G such that $L = L(G)$.

Example: The CFG for balanced parentheses can be defined by following production rules:

S → SS
S → (S)
S → ()

There are two terminal symbols (and) and one nonterminal symbol S. The first rule allows S's to multiply, the second rule allows S's to become enclosed by matching parentheses, and the third rule terminates the recursion.

Starting with S, applying the rules, one can construct:

S → SS → SSS → (S)SS → ((S))SS → ((SS))S(S)
→ (((S))S(S) → ((()()))S(S) → ((()()))()(S)
→ ((()()))()(())

<u>Regular Grammar (Type 1):</u>

In regular grammars, the left hand side is only a single nonterminal symbol, the right side may be the empty string, or a single terminal symbol, or a single terminal symbol followed by a nonterminal symbol, but nothing else. (Sometimes a broader definition is used: one can allow longer strings of terminals or single nonterminals without anything else, making languages easier to denote while still defining the same class of languages.)

A regular grammar can be either right regular or left regular. A **right regular grammar** (also called right linear grammar) is a formal grammar (*N*, Σ, *P*, *S*) such that all the production rules in *P* are of one of the following forms:

1. $B \rightarrow a$ - where *B* is a non-terminal in *N* and *a* is a terminal in Σ
2. $B \rightarrow aC$ - where *B* and *C* are in *N* and *a* is in Σ
3. $B \rightarrow \varepsilon$ - where *B* is in *N* and ε denotes the empty string, i.e. the string of length 0.

Example:

An example of a right regular grammar G with $N = \{S, A\}$, $\Sigma = \{a, b, c\}$, P consists of the following rules

$S \rightarrow aS$
$S \rightarrow bA$
$A \rightarrow \varepsilon$
$A \rightarrow cA$

and S is the start symbol. This grammar describes the same language as the regular expression a*bc*.

Physical Symbol System (PSS):

The **physical symbol system hypothesis** (PSSH), first formulated by **Newell and Simon**, states that **"a physical symbol system [such as a digital computer, for example] has the necessary and sufficient means for intelligent action." The hypothesis implies that**

computers, when we provide them with the appropriate symbol-processing programs, will be capable of intelligent action. It also implies, as Newell and Simon wrote, that "the symbolic behavior of man arises because he has the characteristics of a physical symbol system."

This claim implies both that human thinking is a kind of symbol manipulation (because a symbol system is necessary for intelligence) and that machines can be intelligent (because a symbol system is sufficient for intelligence).

A PSS consists of

- Symbols - set of entities that are physical patterns
- Symbol Structures - number of instances/tokens related in some physical way
- Processes – operate on these expressions to produce other expressions

A natural question to ask about symbols and representation is *what is a symbol?* **Allen Newell** considered this question in *Unified Theories of Cognition*. He differentiated between *symbols* (the phenomena in the abstract) and *tokens* (their physical instantiations). Tokens "stood for" some larger concept. They could be manipulated locally until the information in the larger concept was needed, when local processing would have to stop and access the distal site where the information was stored. The distal information may itself be symbolically encoded, potentially leading to a graph of distal accesses for information.

Newell defined *symbol systems* according to their characteristics. Firstly, they may form a universal computational system. They have

- *memory* to contain the distal symbol information,
- *symbols* to provide a pattern to match or index distal information,
- *operations* to manipulate symbols,
- *interpretation* to allow symbols to specify operations, and,
- *capacities* for:
    1. sufficient memory,
    2. composability (that the operators may make any symbol structure),
    3. interpretability (that symbol structures be able to encode any meaningful arrangement of operations).

Finally, Newell defined symbolic architectures as *the fixed structure that realizes a symbol system*. The fixity implies that the behavior of structures on top of it (i.e. "programs") mainly depend upon the details of the symbols, operations and interpretations at the symbol system level, not upon how the symbol system (and its components) are implemented. How well this ideal hold is a measure of the *strength* of that level.

Here is Simon's own description:

A physical symbol system holds a set of entities, called symbols. These are physical patterns (e.g., chalk marks on a blackboard) that can occur as components of symbol structures. In the case of computers, a symbol system also possesses a number of simple

processes that operate upon symbol structures - processes that create, modify, copy and destroy symbols. A physical symbol system is a machine that, as it moves through time, produces an evolving collection of symbol structures. Symbol structures can, and commonly do, sever as internal representations (e.g., "mental images") of the environment to which the symbol system is seeking to adapt. They allow it to model that environment with greater or less veridicality and in greater or less detail, and consequently to reason about it. Of course, for this capability to be of any use to the symbol system, it must have windows on the world and hands, too. It must have means for acquiring information from the external environment that can be encoded into internal symbols, as well as means for producing symbols that initiate action upon the environment. Thus it must use symbols to designate objects and relations and actions in the world external to the system,

Symbols may also designate processes that the symbol system can interpret and execute. Hence the program that governs the behavior of a symbol system can be stored, along with other symbol structures, in the system's own memory, and executed when activated.

Examples of Symbol Systems

System	**Symbol**	**Expressions**	**Processes**
Logic	And, Or, Not,	Propositions (T/F)	Rules of Inference
Algebra	+, -, *, /, y, z, 1, 2, 3….	Equations (2+3=5)	Rules of Algebra
Digital Computer	0, 1	00000111000…..	Program
Chess	Chess Pieces	Position of pieces on the board	Legal Chess Moves
Brain	Encoded in brain	Thoughts	Mental operations like thinking

The advantages of *symbolic* architectures are:

1. much of human knowledge is symbolic, so encoding it in a computer is more straight-forward;
2. how the architecture reasons may be analogous to how humans do, making it easier for humans to understand;
3. they may be made computationally complete (e.g. Turing Machines).

These advantages have been considered as one of the fundamental tenets of artificial intelligence known as the **physical symbol system hypothesis**. The hypothesis proposes that a physical symbol system has the necessary and sufficient means for general intelligence.