Introduction to the Theory of Computation

Jingde Cheng

Saitama University

An Introduction to the Theory of Computation

- Enumerability and Diagonalization
- Finite Automata and Regular Languages
- Pushdown Automata and Context-Free Languages
- Computation: Turing Machines
- Computation: Turing-Computability (Turing-Decidability)
- Computation: Reducibility and Turing-Reducibility
- Computation: Recursive Functions
- Computation: Recursive Sets and Relations
- Equivalent Definitions of Computability
- Advanced Topics in Computability Theory
- Computational Complexity
- Time Complexity
- Space Complexity
- Intractability
- Advanced Topics in Complexity Theory

Coding Turing Machine Computations [BBJ-ToC-07]

♣ Coding TM computations

- Let $F(m, x)$ = otpt(m, x, halt(m, x)), a recursive function, the result of coding TM computation. $F(m, x)$ will be the value of the function computed by the TM with code number m for argument x, if that function is defined for that argument, and will be undefined otherwise.
- If f is a Turing-computable function, then for some m -- namely, for the code number of any Turing machine computing f -- we have $f(x) = F(m, x)$ for all x. Since F is recursive, it follows that f is recursive.

♣ Theorem

- A function is recursive IFF it is Turing computable (Turing-decidable).

Universal Functions and Universal Turing Machines [BBJ-ToC-07]

♣ Universal functions

- [D] An $(n+1)$-place recursive function F with the property that for every n-place recursive function f there is an m such that $f(x_1, ..., x_n) = F(m, x_1, ..., x_n)$ is called a *universal function*.

♣ Theorem (Existence theorem of universal functions)

- For every k there exists a universal k-place recursive function (whose graph relation is semi-recursive).

♣ Universal Turing machines

- [D] A Turing machine for computing a universal function is called a *universal Turing machine*.
- If U is such a machine (for, say, $k = 1$), then for any TM M we like, the value computed by M for a given argument x will also be computed by U given a code m for M as a further argument in addition to x.

Corollaries of the Existence Theorem of Universal Functions [BBJ-ToC-07]

♣ Corollary (Second graph principle)

- The graph relation of a recursive function is semi-recursive.

♣ Corollary

- Let A be a set of natural numbers. Then the following conditions are equivalent:
  (a) A is the range of some recursive total or partial function.
  (b) A is the domain of some recursive total or partial function.
  (c) A is semi-recursive.

♣ Corollary

- There exists a recursively enumerable set that is not recursive.

Historical Notes [BBJ-ToC-07]

♣ Universal TMs as the theoretical background for general-purpose, programmable computers

- Historically, the theory of Turing computability (including the proof of the existence of a universal TM) was established before (indeed, a decade or more before) the age of general-purpose, programmable computers, and in fact formed a significant part of the theoretical background for the development of such computers.

♣ Universal TMs as the first theoretical assurance for general-purpose, programmable computers

- We can now say more specifically that the theorem that there exists a universal TM, together with Turing's thesis that all effectively computable functions are Turing computable, heralded the arrival of the computer age by giving the first theoretical assurance that in principle a general-purpose computer could be designed that could be made to mimic any special-purpose computer desired, simply by giving it coded instructions as to what machine it is to mimic as an additional input along with the arguments of the function we want computed.

The Universal Turing Machine [L-ToC-17]

♣ **The universal (reprogrammable) TM [Turing, 1936]**

- **[D] The *universal (reprogrammable) Turing machine* M_u is a TM that, given as input the description of any TM M and a string w, can simulate the computation of M on w.**

♣ **Q and Γ of M_u**

- **We assume that Q = $\{q_1, q_2, ..., q_n\}$, with q_1 the initial state, q_2 the single final state, and Γ = $\{a_1, a_2, ..., a_m\}$, where a_1 represents the blank.**
- **We then select an encoding in which q_1 is represented by 1, q_2 is represented by 11, and so on. Similarly, a_1 is encoded as 1, a_2 as 11, etc, L is encoded as 1, R as 11. The symbol 0 will be used as a separator between the 1's.**

The Universal Turing Machine [L-ToC-17]

♣ **The transition function δ of M_u**

- **With the initial and final state and the blank defined by this convention, any TM can be described completely with transition function δ only.**
- **The transition function is encoded according to this scheme, with the arguments and result in some prescribed sequence.**
- **For example, $\delta(q_1, a_2) = (q_2, a_3, \mathrm{L})$ might appear as $\cdots q_1 0 a_2 0 q_2 0 a_3 0 \mathrm{L} 0 \cdots$, $\cdots 10110110111010 \cdots$.**
- **It follows from this that any TM M has a finite encoding as a string on $\{0, 1\}^+$ and that, given any encoding of M, we can decode it uniquely.**

♣ **Notes**

- **The observation that every TM can be represented by a string of 0's and 1's has important implications.**
- **The set of all TMs, although infinite, is countable.**

The Universal Turing Machine [L-ToC-17]

♣ **Structure of M_u**

- **A universal TM M_u then has an input alphabet that includes {0, 1} and the structure of a multi-tape TM.**

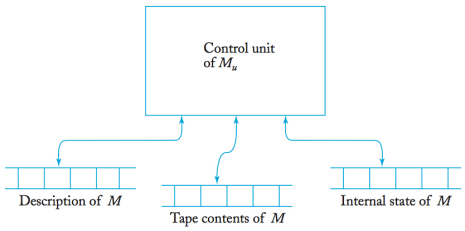


FIGURE 10.16

The Universal Turing Machine [L-ToC-17]

♣ **Simulating the computation of M on w by M_u**

- **For any input M and w, tape 1 will keep an encoded definition of M. Tape 2 will contain the tape contents of M, and tape 3 the internal state of M.**
- **M_u looks first at the contents of tapes 2 and 3 to determine the configuration of M.**
- **It then consults tape 1 to see what M would do in this configuration.**
- **Finally, tapes 2 and 3 will be modified to reflect the result of the move.**

The Universal Turing Machine [W-ToC-12]

3.5. A Universal Turing Machine

From the enumeration of all Turing machines and the pairing function we can define a *universal* Turing machine U; that is, a machine that will emulate *every* other machine. Simply set

$$U(\langle e, x\rangle) = \varphi_e(x).$$

U decodes the single number it receives into a pair (e, x), decodes e into the appropriate set of quadruples, and uses x as input, acting according to the quadruples it decoded. This procedure is computable because decoding the pairing function, decoding Turing machines from indices, and executing quadruples on a given input are computable procedures. Turing [85] gives an explicit, full construction of a universal machine.

Note that of course there are infinitely many universal Turing machines, as there are for any program via padding, and that a universal machine exists for any collection of functions that may be indexed. Although we will use U and analogously defined universal machines for other indexings in this section, typically we simply refer to the individual indexed functions, using $\varphi_e(x)$ instead of $U(\langle e, x\rangle)$.

The Universal Turing Machine [HS-ToC-11]

Since every word is a number and vice versa, we may think of a Turing-machine code as a number. The code for a Turing machine M is called the *Gödel number* of M. If e is a Gödel number, then M_e is the Turing machine whose Gödel number is e.

Let U be a Turing machine that computes on input e and x and that implements the following algorithm:

if e is a code
  then simulate M_e on input x
  else output 0.

(Why are you convinced that a Turing machine with this behavior exists?) U is a *universal* Turing machine. To put it differently, U is a general-purpose, stored-program computer: U accepts as input two values: a "stored program" e, and "input to e," a word x. If e is the correct code of a program M_e, then U computes the value of M_e on input x.

Early computers had their programs hard-wired into them. Several years after Turing's 1936 paper, von Neumann and co-workers built the first computer that stored instructions internally in the same manner as data. Von Neumann knew Turing's work, and it is believed that von Neumann was influenced by Turing's universal machine. Turing's machine U is the first conceptual general-purpose, stored-program computer.

The Universal Turing Machine [F-ToC-09]

2.4.2 The universal Turing machine

It is possible to give a code for each Turing machine, so that from the code we can retrieve the machine. An easy way of doing this is as follows.

Assume the machine has n states $q_0, \ldots, q_{n-1}$, where each q_i is a number, q_0 is the initial state, and the last m states are final. Also assume that the input alphabet is $\{1\}$ and the tape alphabet is $\{0, 1\}$ (where 0 plays the role of a blank symbol). It is well known that a binary alphabet is sufficient to encode any kind of data, so there is no loss of generality in making this assumption. The transition function can be represented as a table, or equivalently a list of 5-tuples of the form (q, s, q', s', d), where q represents the current state, s the symbol on the tape under the head, q' the new state, s' the symbol written on the tape, and d the direction of movement, which we will write as $0, 1$. The order of the tuples is not important here. Thus, we can assume without loss of generality that the transition function is represented by a list l of tuples. The full description of the machine under these assumptions is given by the tuple (n, m, l), where l is the list representing the transition function, n the number of states, and m the number of final states, as indicated above. We will say that the tuple is the *code* for the machine since from it we can recover the original machine. In fact, the code for the machine is not unique since we can reorder the list l and still obtain an equivalent machine.

Now we can see the codes of Turing machines as words, and as such they can be used as input for a Turing machine. It is then possible to define a Turing machine U such that, when the code of a machine A is written on the tape, together with an input word w for A, U decodes it and simulates the behaviour of the machine A on w. The machine U is usually called the *universal Turing machine.*

The Universal Turing Machine [LP-ToC-98]

Let $M = (K, \Sigma, \delta, s, H)$ be a Turing machine, and let i and j be the smallest integers such that $2^i \geq |K|$, and $2^j \geq |\Sigma| + 2$. Then each state in K will be represented as a q followed by a binary string of length i; each symbol in Σ will be likewise represented as the letter a followed by a string of j bits. The head directions $\leftarrow$ and $\rightarrow$ will also be treated as "honorary tape symbols" (they were the reason for the "+2" term in the definition of j). We fix the representations of the special symbols $\sqcup, \triangleright, \leftarrow$, and $\rightarrow$ to be the lexicographically four smallest symbols, respectively: $\sqcup$ will always be represented as $a0^j$, $\triangleright$ as $a0^{j-1}1$, $\leftarrow$ as $a0^{j-2}10$, and $\rightarrow$ as $a0^{j-2}11$. The start state will always be represented as the lexicographically first state, $q0^i$. Notice that we require the use of leading zeros in the strings that follow the symbols a and q, to bring the total length to the required level.

We shall denote the representation of the whole Turing machine M as "M". "M", consists of the transition table δ. That is, it is a sequence of strings of the form (q, a, p, b), with q and p representations of states and a, b of symbols, separated by commas and included in parentheses. We adopt the convention that the quadruples are listed in *increasing lexicographic order*, starting with $\delta(s, \sqcup)$. The set of halting states H will be determined indirectly, by the absence of its states as first components in any quadruple of "M". If M decides a language, and thus $H = \{y, n\}$, we will adopt the convention that y is the lexicographically smallest of the two halt states.

This way, any Turing machine can be represented. We shall use the same method to represent *strings* in the alphabet of the Turing machine. Any string $w \in \Sigma^*$ will have a unique representation, also denoted "w", namely, the juxtaposition of the representations of its symbols.

Grammars [L-ToC-17]

♣Formal definition of grammars

DEFINITION 1.1

A grammar G is defined as a quadruple

$$G = (V, T, S, P),$$

where V is a finite set of objects called **variables**,
T is a finite set of objects called **terminal symbols**,
$S \in V$ is a special symbol called the **start** variable,
P is a finite set of **productions**.

It will be assumed without further mention that the sets V and T are non-empty and disjoint.

♣Languages generated by grammars

DEFINITION 1.2

Let $G = (V, T, S, P)$ be a grammar. Then the set

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

is the language generated by G.

Grammars [L-ToC-17]

♣Production rules

- **The *production rules* are the heart of a grammar; they specify how the grammar transforms one string into another, and through this they define a language associated with the grammar.**
- **All production rules are of the form $x \rightarrow y$, where x is an element of $(V \cup T)^+$ and y is in $(V \cup T)^*$.**

♣Applications of production rules

- **The productions are applied in the following manner: Given a string w of the form $w = uxv$, we say the production $x \rightarrow y$ is applicable to this string, and we may use it to replace x with y, thereby obtaining a new string $z = uyv$.**
- **This is written as $w \Rightarrow z$. We say that w derives z or that z is derived from w.**

Grammars [L-ToC-17]

♣Applications of production rules

- **Successive strings are derived by applying the productions of the grammar in arbitrary order.**
- **A production can be used whenever it is applicable, and it can be applied as often as desired.**
- **If $w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n$, we say that w_1 derives w_n and write $w_1 \Rightarrow^* w_n$. The $*$ indicates that an unspecified number of steps (including zero) can be taken to derive w_n from w_1.**
- **By applying the production rules in a different order, a given grammar can normally generate many strings.**
- **The set of all such terminal strings is the language defined or generated by the grammar.**

Grammar Examples [L-ToC-17]

EXAMPLE 1.11

Consider the grammar

$$G = (\{S\}, \{a, b\}, S, P),$$

with P given by

$$S \rightarrow aSb,$$
$$S \rightarrow \lambda.$$

Then

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb,$$

so we can write

$$S \overset{*}{\Rightarrow} aabb.$$

The string $aabb$ is a sentence in the language generated by G, while $aaSbb$ is a sentential form.

A grammar G completely defines $L(G)$, but it may not be easy to get a very explicit description of the language from the grammar. Here, however, the answer is fairly clear. It is not hard to conjecture that

$$L(G) = \{a^n b^n : n \geq 0\},$$

and it is easy to prove it. If we notice that the rule $S \rightarrow aSb$ is recursive, a proof by induction readily suggests itself. We first show that all sentential forms must have the form

$$w_k = a^k S b^k. \qquad (1.7)$$

Suppose that (1.7) holds for all sentential forms w_k of length $2i + 1$ or less. To get another sentential form (which is not a sentence), we can only apply the production $S \rightarrow aSb$. This gets us

$$a^k S b^k \Rightarrow a^{k+1} S b^{k+1},$$

so that every sentential form of length $2i + 3$ is also of the form (1.7). Since (1.7) is obviously true for $i = 1$, it holds by induction for all i. Finally, to get a sentence, we must apply the production $S \rightarrow \lambda$, and we see that

$$S \overset{*}{\Rightarrow} a^n S b^n \Rightarrow a^n b^n$$

represents all possible derivations. Thus, G can derive only strings of the form $a^n b^n$.

We also have to show that all strings of this form can be derived. This is easy; we simply apply $S \rightarrow aSb$ as many times as needed, followed by $S \rightarrow \lambda$.

Grammar Examples [L-ToC-17]

EXAMPLE 1.12

Find a grammar that generates

$$L = \{a^n b^{n+1} : n \geq 0\}.$$

The idea behind the previous example can be extended to this case. All we need to do is generate an extra b. This can be done with a production $S \to Ab$, with other productions chosen so that A can derive the language in the previous example. Reasoning in this fashion, we get the grammar $G = (\{S, A\}, \{a, b\}, S, P)$, with productions

$$S \to Ab,$$
$$A \to aAb,$$
$$A \to \lambda.$$

Derive a few specific sentences to convince yourself that this works.

Unrestricted Grammars [L-ToC-17]

♣ **Unrestricted grammars**

- **In an unrestricted grammar, essentially no conditions are imposed on the productions.**
- **Any number of variables and terminals can be on the left or right, and these can occur in any order.**
- **There is only one restriction: λ or ε (the empty string) is not allowed as the left side of a production.**

♣ **Unrestricted grammars vs. restricted grammars**

- **Unrestricted grammars are much more powerful than restricted forms like the regular and context-free grammars.**
- **In fact, unrestricted grammars correspond to the largest family of languages so we can hope to recognize by mechanical means; that is, unrestricted grammars generate exactly the family of recursively enumerable languages.**

Unrestricted Grammars [L-ToC-17]

DEFINITION 11.3

A grammar $G = (V, T, S, P)$ is called **unrestricted** if all the productions are of the form

$$u \to v,$$

where u is in $(V \cup T)^+$ and v is in $(V \cup T)^*$.

THEOREM 11.6

Any language generated by an unrestricted grammar is recursively enumerable.

THEOREM 11.7

For every recursively enumerable language L, there exists an unrestricted grammar G, such that $L = L(G)$.

Context-sensitive Grammars and Languages [L-ToC-17]

♣ **Context-sensitive grammars**

- **Between the restricted, context-free grammars and the general, unrestricted grammars, a great variety of "somewhat restricted" grammars can be defined.**
- **Context-sensitive grammars generate languages associated with a restricted class of Turing machines, linear bounded automata.**

♣ **"Context-sensitive" grammars**

- **All context-sensitive grammars can be rewritten in a normal form in which all productions are of the form *xAy* → *xvy*.**
- **This is equivalent to saying that the production *A* → *v* can be applied only in the situation where *A* occurs in a context of the string *x* on the left and the string *y* on the right.**

Context-sensitive Grammars and Languages [L-ToC-17]

DEFINITION 11.4

A grammar $G = (V, T, S, P)$ is said to be **context sensitive** if all productions are of the form

$$x \to y,$$

where $x, y \in (V \cup T)^+$ and

$$|x| \leq |y|. \qquad (11.15)$$

DEFINITION 11.5

A language L is said to be context sensitive if there exists a context-sensitive grammar G, such that $L = L(G)$ or $L = L(G) \cup \{\lambda\}$.

Context-sensitive Grammars and Languages [L-ToC-17]

THEOREM 11.8

For every context-sensitive language L not including λ, there exists some linear bounded automaton M such that $L = L(M)$.

THEOREM 11.9

If a language L is accepted by some linear bounded automaton M, then there exists a context-sensitive grammar that generates L.

THEOREM 11.10

Every context-sensitive language L is recursive.

THEOREM 11.11

There exists a recursive language that is not context sensitive.

The Chomsky Hierarchy [L-ToC-17]

♣ **Various languages**

- L_{RE}: the recursively enumerable languages
- L_{CS}: the context-sensitive languages
- L_{CF}: the context-free languages
- L_{REG}: the regular languages

♣ **The Chomsky hierarchy**

- One way of exhibiting the relationship between these families of language is by the Chomsky hierarchy.
- Noam Chomsky, a founder of formal language theory, provided an initial classification into four language types, type 0 to type 3.

The Chomsky Hierarchy [L-ToC-17]

♣ **The Chomsky hierarchy**

- Noam Chomsky, a founder of formal language theory, provided an initial classification into four language types, type 0 to type 3.
- This original terminology has persisted and one finds frequent references to it, but the numeric types are actually different names for the language families we have studied.
- Type 0 languages are those generated by unrestricted grammars, that is, the recursively enumerable languages.
- Type 1 consists of the context-sensitive languages, type 2 consists of the context-free languages, and type 3 consists of the regular languages.
- As we have seen, each language family of type i is a proper subset of the family of type i−1.

The Chomsky Hierarchy [L-ToC-17]

♣ **The Chomsky hierarchy**

- A diagram (Figure 11.3) exhibits the relationship clearly.
- Figure 11.3 shows the original Chomsky hierarchy.
- We have also met several other language families that can be fitted into this picture.
- Including the families of deterministic context-free languages (L_{DCF}) and recursive languages (L_{REC}), we arrive at the extended hierarchy shown in Figure 11.4.
- The relationship between regular, linear, deterministic context-free, and nondeterministic context-free languages is as shown in Figure 11.5.

The Chomsky Hierarchy [L-ToC-17]

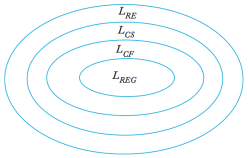


FIGURE 11.3

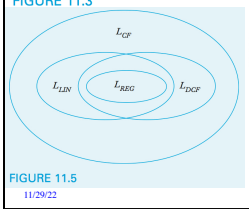


FIGURE 11.5

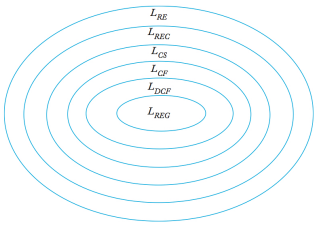


FIGURE 11.4

The Theory of Post Systems: What is It? [L-ToC-17]

♣ **Post systems**

- A Post system looks very much like an unrestricted grammar consisting of an alphabet and some production rules by which successive strings can be derived.
- But there are significant differences in the way in which the productions are applied.

♣ **Post's paper**

- Emil Post, "Formal Reductions of the General Combinatorial Decision Problem," American Journal of Mathematics, 65 (2) 197-215, 1943.

Post Systems and Their Languages [L-ToC-17]

DEFINITION 13.3

A Post system Π is defined by

$$\Pi = (C, V, A, P),$$

where

C is a finite set of constants, consisting of two disjoint sets C_N, called the **nonterminal constants**, and C_T, the set of **terminal constants**,

V is a finite set of variables,

A is a finite set from C^*, called the **axioms**,

P is a finite set of productions.

DEFINITION 13.4

The language generated by the Post system $\Pi = (C, V, A, P)$ is

$$L(\Pi) = \left\{ w \in C_T^* : w_0 \overset{*}{\Rightarrow} w \text{ for some } w_0 \in A \right\}.$$

THEOREM 13.6

A language is recursively enumerable if and only if there exists some Post system that generates it.

Productions in Post Systems [L-ToC-17]

The productions in a Post system must satisfy certain restrictions. They must be of the form

$$x_1V_1x_2\cdots V_nx_{n+1} \rightarrow y_1W_1y_2\cdots W_my_{m+1}, \qquad (13.1)$$

where $x_i,\ y_i \in C^*$, and $V_i,\ W_i \in V$, subject to the requirement that any variable can appear at most once on the left, so that

$$V_i \neq V_j \text{ for } i \neq j,$$

and that each variable on the right must appear on the left, that is,

$$\bigcup_{i=1}^{m} W_i \subseteq \bigcup_{i=1}^{n} V_i.$$

Suppose we have a string of terminals of the form $x_1w_1x_2w_2\cdots w_nx_{n+1}$, where the substrings $x_1,\ x_2\cdots$ match the corresponding strings in (13.1) and $w_i \in C^*$. We can then make the identification $w_1 = V_1,\ w_2 = V_2, \ldots$, and substitute these values for the W's on the right of (13.1). Since every W is some V_i that occurs on the left, it is assigned a unique value, and we get the new string $y_1w_iy_2w_j\cdots y_{m+1}$. We write this as

$$x_1w_1x_2w_2\cdots x_{n+1} \Rightarrow y_1w_iy_2w_j\cdots y_{m+1}.$$

Post Systems: Examples [L-ToC-17]

EXAMPLE 13.5

Consider the Post system with

$$C_T = \{a, b\},$$
$$C_N = \varnothing,$$
$$V = \{V_1\},$$
$$A = \{\lambda\},$$

and production

$$V_1 \rightarrow aV_1b.$$

This allows the derivation

$$\lambda \Rightarrow ab \Rightarrow aabb.$$

In the first step, we apply (13.1) with the identification $x_1 = \lambda,\ V_1 = \lambda,\ x_2 = \lambda,\ y_1 = a,\ W_1 = V_1$, and $y_2 = b$. In the second step, we re-identify $V_1 = ab$, leaving everything else the same. If you continue with this, you will quickly convince yourself that the language generated by this particular Post system is $\{a^nb^n : n \geq 0\}$.

Post Systems: Examples [L-ToC-17]

EXAMPLE 13.6

Consider the Post system with

$$C_T = \{1, +, =\},$$
$$C_N = \varnothing,$$
$$V = \{V_1, V_2, V_3\},$$
$$A = \{1+1=11\},$$

and productions

$$V_1 + V_2 = V_3 \rightarrow V_1 1 + V_2 = V_3 1,$$
$$V_1 + V_2 = V_3 \rightarrow V_1 + V_2 1 = V_3 1.$$

The system allows the derivation

$$1+1 = 11 \Rightarrow 11+1 = 111$$
$$\Rightarrow 11+11 = 1111.$$

Interpreting the strings of 1's as unary representations of integers, the derivation can be written as

$$1+1 = 2 \Rightarrow 2+1 = 3 \Rightarrow 2+2 = 4.$$

The language generated by this Post system is the set of all identities of integer additions, such as $2+2=4$, derived from the axiom $1+1=2$.

Post Systems as a Model of Computation [L-ToC-17]

♣ **Post systems as a model of computation**

- **Example 13.6 illustrates in a simple manner the original intent of Post systems as a mechanism for rigorously proving mathematical statements from a set of axioms.**
- **It also shows the inherent awkwardness of such a completely rigorous approach and why it is rarely used.**
- **But Post systems, even though they are cumbersome for proving complicated theorems, are general models for computation.**

♣ **Post's paper**

- **Emil Post, "Formal Reductions of the General Combinatorial Decision Problem," American Journal of Mathematics, 65 (2) 197-215, 1943.**

Post Systems as a Model of Computation [L-ToC-17]

THEOREM 13.6

A language is recursively enumerable if and only if there exists some Post system that generates it.

Proof: The arguments here are relatively simple and we sketch them briefly. First, since a derivation by a Post system is completely mechanical, it can be carried out on a Turing machine. Therefore, any language generated by a Post system is recursively enumerable.

For the converse, remember that any recursively enumerable language is generated by some unrestricted grammar G, having productions all of the form

$$x \rightarrow y,$$

with $x,\ y \in (V \cup T)^*$. Given any unrestricted grammar G, we create a Post system $\Pi = (V_\Pi, C, A, P_\Pi)$, where $V_\Pi = \{V_1, V_2\}, C_N = V, C_T = T, A = \{S\}$, and with productions

$$V_1xV_2 \rightarrow V_1yV_2,$$

for every production $x \rightarrow y$ of the grammar. It is then an easy matter to show that a w can be generated by the Post system Π if and only if it is in the language generated by G. ■

Rewriting Systems: What are They? [L-ToC-17]

♣ **Various grammars, TMs, and Post systems**

- **Various grammars have a number of things in common with Post systems: They are all based on an alphabet in which strings are written, and some rules by which one string can be obtained from another.**
- **Even a TM can be viewed this way, since its instantaneous description is a string that completely defines its configuration. The program is then just a set of rules for producing one such string from a previous one.**

♣ **Rewriting systems**

- **The above observations can be formalized in the concept of a rewriting system.**

Rewriting Systems: What are They? [L-ToC-17]

♣ Rewriting systems

- [D] Generally, a rewriting system consists of an alphabet Σ and a set of rules or productions by which a string in Σ^+ can produce another.
- What distinguishes one rewriting system from another is the nature of Σ and restrictions for the application of the productions.

♣ Notes

- The idea of a rewriting system is quite broad and allows any number of specific cases.
- Rewriting systems provide a general model for computation and have many applications in CS.

Matrix Grammars [L-ToC-17]

♣ Matrix grammars

- [D] *Matrix grammars* differ from the grammars we have previously studied (which are often called *phrase-structure grammars*) in how the productions can be applied.
- [D] For matrix grammars, the set of productions consists of subsets $P_1, P_2, ..., P_n$, each of which is an ordered sequence $x_1 \rightarrow y_1, x_2 \rightarrow y_2, ...$
- [D] Whenever the first production of some set P_i is applied, we must next apply the second one to the string just created, then the third one, and so on.
- [D] We cannot apply the first production of P_i unless all other productions in this set can also be applied.

Matrix Grammars: An Example [L-ToC-17]

EXAMPLE 13.7

Consider the matrix grammar

$$P_1 : S \rightarrow S_1S_2,$$
$$P_2 : S_1 \rightarrow aS_1, S_2 \rightarrow bS_2c,$$
$$P_3 : S_1 \rightarrow \lambda, S_2 \rightarrow \lambda.$$

A derivation with this grammar is

$$S \Rightarrow S_1S_2 \Rightarrow aS_1bS_2c \Rightarrow aaS_1bbS_2cc \Rightarrow aabbcc.$$

Note that whenever the first rule of P_2 is used to create an a, the second one also has to be used, producing a corresponding b and c. This makes it easy to see that the set of terminal strings generated by this matrix grammar is

$$L = \{a^n b^n c^n : n \geq 0\}.$$

The Power of Matrix Grammars [L-ToC-17]

♣ The power of matrix grammars

- Matrix grammars contain phrase-structure grammars as a special case in which each P_i contains exactly one production.
- Also, since matrix grammars represent algorithmic processes, they are governed by Church-Turing thesis.
- We conclude from this that matrix grammars and phrase-structure grammars have the same power as models of computation.
- But, as Example 13.7 shows, sometimes the use of a matrix grammar gives a much simpler solution than we can achieve with an unrestricted phrase-structure grammar.

Markov Algorithms [L-ToC-17]

♣ Markov algorithms

- [D] A *Markov algorithm* is a rewriting system whose productions $x \rightarrow y$ are considered ordered.
- [D] In a derivation, the first applicable production must be used.
- [D] Furthermore, the leftmost occurrence of the substring x must be replaced by y.
- [D] Some of the productions may be singled out as terminal productions; they will be shown as $x \rightarrow \cdot y$.

Markov Algorithms [L-ToC-17]

♣ Derivation

- A derivation starts with some string $w \in \Sigma^*$ and continues either until a terminal production is used or until there are no applicable productions.
- For language acceptance, a set $T \subseteq \Sigma$ of terminals is identified.
- Starting with a terminal string, productions are applied until the empty string is produced.

Markov Algorithms [L-ToC-17]

DEFINITION 13.5

Let M be a Markov algorithm with alphabet Σ and terminals T. Then the set

$$L(M) = \left\{ w \in T^* : w \overset{*}{\Rightarrow} \lambda \right\}$$

is the language accepted by M.

THEOREM 13.7

A language is recursively enumerable if and only if there exists a Markov algorithm for it.

Markov Algorithms: Examples [L-ToC-17]

EXAMPLE 13.8

Consider the Markov algorithm with $\Sigma = T = \{a, b\}$ and productions

$$ab \rightarrow \lambda,$$
$$ba \rightarrow \lambda.$$

Every step in the derivation annihilates a substring ab or ba, so

$$L(M) = \left\{ w \in \{a,b\}^* : n_a(w) = n_b(w) \right\}.$$

- **$n_a(w)$ and $n_b(w)$ denote the number of *a*'s and *b*'s in the string *w*.**

Markov Algorithms: Examples [L-ToC-17]

EXAMPLE 13.9

Find a Markov algorithm for

$$L = \{a^n b^n : n \geq 0\}.$$

An answer is

$$ab \rightarrow S,$$
$$aSb \rightarrow S,$$
$$S \rightarrow \cdot\lambda.$$

If in this last example we take the first two productions and reverse the left and right sides, we get a context-free grammar that generates the language L. In a certain sense, Markov algorithms are simply phrase-structure grammars working backward. This cannot be taken too literally, since it is not clear what to do with the last production. But the observation does provide a starting point for a proof of the following theorem that characterizes the power of Markov algorithms.

L-Systems [L-ToC-17]

♣ **L-systems**

- **[D] L-systems (developed by A. Lindenmayer) are essentially parallel rewriting systems.**
- **In each step of a derivation, every symbol has to be rewritten.**
- **The productions of an L-system must be of the form $a \rightarrow u$, where $a \in \Sigma$ and $u \in \Sigma^*$.**
- **When a string is rewritten, one such production must be applied to every symbol of the string before the new string is generated.**

L-Systems: An Example [L-ToC-17]

EXAMPLE 13.10

Let $\Sigma = \{a\}$ and

$$a \rightarrow aa$$

define an L-system. Starting from the string a, we can make the derivation

$$a \Rightarrow aa \Rightarrow aaaa \Rightarrow aaaaaaaa.$$

The set of strings so derived is clearly

$$L = \left\{ a^{2^n} : n \geq 0 \right\}.$$

The Power of L-Systems [L-ToC-17]

♣ **The power of L-systems**

- **L-systems with productions of the form $a \rightarrow u$ are not sufficiently general to provide for all algorithmic computations.**

♣ **An extension of L-systems**

- **An extension of the idea provides the necessary generalization.**
- **In an extended L-system, productions are of the form $(x, a, y) \rightarrow u$, where $a \in \Sigma$ and $x, y, u \in \Sigma^*$, with the interpretation that a can be replaced by u only if it occurs as part of the string xay.**
- **It is known that such extended L-systems are general models of computation.**

Introduction to the Theory of Computation

- Enumerability and Diagonalization
- Finite Automata and Regular Languages
- Pushdown Automata and Context-Free Languages
- Computation: Turing Machines
- Computation: Turing-Computability (Turing-Decidability)
- Computation: Reducibility and Turing-Reducibility
- Computation: Recursive Functions
- Computation: Recursive Sets and Relations
- Equivalent Definitions of Computability
- Advanced Topics in Computability Theory
- Computational Complexity
- Time Complexity
- Space Complexity
- Intractability
- Advanced Topics in Complexity Theory

Self-Reference TM *SELF* [S-ToC-13]

♣ Self-reference TM *SELF*

- [D] *SELF* is a TM such that it ignores its input and prints out a copy of its own description.
- To help describe *SELF*, we need the following lemma.

♣ Lemma

- There is a computable function q: $\Sigma^* \rightarrow \Sigma^*$, where if w is any string, $q(w)$ is the description of a TM P_w that prints out w and then halts.

♣ Composition of self-reference TM *SELF*

- The TM *SELF* is in two parts: *A* and *B*. We think of *A* and *B* as being two separate procedures that go together to make up *SELF*.
- We want *SELF* to print out <*SELF*> = <*AB*>.

Self-Reference TM *SELF* [S-ToC-13]

LEMMA 6.1

There is a computable function q: $\Sigma^* \longrightarrow \Sigma^*$, where if w is any string, $q(w)$ is the description of a Turing machine P_w that prints out w and then halts.

PROOF Once we understand the statement of this lemma, the proof is easy. Obviously, we can take any string w and construct from it a Turing machine that has w built into a table so that the machine can simply output w when started. The following TM Q computes $q(w)$.

Q = "On input string w:

1. Construct the following Turing machine P_w.
   P_w = "On any input:
   1. Erase input.
   2. Write w on the tape.
   3. Halt."
2. Output $\langle P_w \rangle$."

Construction of Self-Reference TM *SELF* [S-ToC-13]

♣ The jobs of part *A* and part *B*

- Part *A* runs first and upon completion passes control to *B*.
- The job of *A* is to print out a description of *B*, and conversely the job of *B* is to print out a description of *A*.
- The result is the desired description of *SELF*.
- The jobs are similar, but they are carried out differently.

♣ Part A

- For *A* we use the machine $P_{<B>}$, described by $q(<B>)$, which is the result of applying the function q to <*B*>. Thus, part *A* is a TM that prints out <*B*>.
- Our description of *A* depends on having a description of *B*. So we cannot complete the description of *A* until we construct *B*.

Construction of Self-Reference TM *SELF* [S-ToC-13]

♣ Part *B*

- Note: We cannot define *B* with *q*(<*A*>). Doing so would define *B* in terms of *A*, which in turn is defined in terms of *B*. That would be a circular definition of an object in terms of itself, a logical transgression.
- We define *B* so that it prints *A* by using a different strategy: *B* computes *A* from the output that *A* produces.
- We defined <*A*> to be *q*(<*B*>).
- If B can obtain <*B*>, it can apply *q* to that and obtain <*A*>. But how does *B* obtain <*B*>? It was left on the tape when *A* finished!
- So *B* only needs to look at the tape to obtain <*B*>.
- Then after *B* computes *q*(<*B*>) = <*A*>, it combines *A* and *B* into a single machine and writes its description <*AB*> = <*SELF*> on the tape.

Construction of Self-Reference TM *SELF* [S-ToC-13]

$A = P_{\langle B \rangle}$, and

B = "On input $\langle M \rangle$, where M is a portion of a TM:

1. Compute $q(\langle M \rangle)$.
2. Combine the result with $\langle M \rangle$ to make a complete TM.
3. Print the description of this TM and halt."

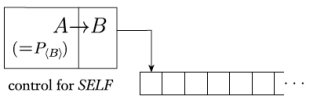


FIGURE 6.2
Schematic of *SELF*, a TM that prints its own description

Running Self-Reference TM *SELF* [S-ToC-13]

♣ **Running *SELF***

- If we now run *SELF*, we observe the following behavior:
  1. First *A* runs. It prints <*B*> on the tape.
  2. *B* starts. It looks at the tape and finds its input, <*B*>.
  3. *B* calculates *q*(<*B*>) = <*A*>, and combines that with <*B*> into a TM description, <*SELF*>.
  4. *B* prints this description and halts.

♣ **Implementation of SELF**

- We can easily implement this construction in any programming language to obtain a program that outputs a copy of itself.

Self-Reference: English Example [S-ToC-13]

- Suppose that we want to give an English sentence that commands the reader to print a copy of the same sentence. One way to do so is to say: Print out this sentence.
- This sentence has the desired meaning because it directs the reader to print a copy of the sentence itself.
- Consider the following alternative.
  Print out two copies of the following, the second one in quotes:
  "Print out two copies of the following, the second one in quotes:"
- In this sentence, the self-reference is replaced with the same construction used to make the TM *SELF*.
  Part *B* of the construction is the clause:
  Print out two copies of the following, the second one in quotes:
  Part *A* is the same, with quotes around it. *A* provides a copy of *B* to *B* so *B* can process that copy as the TM does.

The Recursion Theorem [S-ToC-13]

♣ **The role of the recursion theorem**

- The recursion theorem provides the ability to implement the self-referential "this" into any programming language.
- With it, any program has the ability to refer to its own description, which has certain applications.

♣ **Note**

- The recursion theorem extends the technique we used in constructing *SELF* so that a program can obtain its own description and then go on to compute with it, instead of merely printing it out.

The Recursion Theorem [S-ToC-13]

♣ **The recursion theorem**

THEOREM **6.3**

Recursion theorem Let T be a Turing machine that computes a function $t\colon \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$. There is a Turing machine R that computes a function $r\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$r(w) = t(\langle R \rangle, w).$$

♣ **What the recursion theorem means?**

- To make a TM that can obtain its own description and then compute with it, we need only make a machine *T*, that compute function *t* which receives the description <*T*> of the machine *T* as an extra input.
- Then we can produce a machine *R*, that compute function *r*, which operates exactly as *T* does but with *R*'s description filled in automatically.

The Recursion Theorem [S-ToC-13]

PROOF The proof is similar to the construction of *SELF*. We construct a TM R in three parts, A, B, and T, where T is given by the statement of the theorem; a schematic diagram is presented in the following figure.

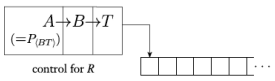


FIGURE **6.4**
Schematic of R

Here, A is the Turing machine $P_{\langle BT \rangle}$ described by $q(\langle BT \rangle)$. To preserve the input w, we redesign q so that $P_{\langle BT \rangle}$ writes its output following any string preexisting on the tape. After A runs, the tape contains $w\langle BT \rangle$.

Again, B is a procedure that examines its tape and applies q to its contents. The result is $\langle A \rangle$. Then B combines A, B, and T into a single machine and obtains its description $\langle ABT \rangle = \langle R \rangle$. Finally, it encodes that description together with w, places the resulting string $\langle R, w \rangle$ on the tape, and passes control to T.

Terminology for the Recursion Theorem [S-ToC-13]

♣ **The implication of the recursion theorem**

- The recursion theorem states that TMs can obtain their own description and then go on to compute with it. The theorem is a handy tool for solving certain problems concerning the theory of algorithms.

♣ **Using the recursion theorem**

- We can use the recursion theorem in the following way when designing TM algorithms. If we are designing a TM *M*, we can include the phrase "obtain own description <*M*> " in the informal description of *M*'s algorithm.
- Upon having obtained its own description, *M* can then go on to use it as it would use any other computed value. *M* might simply print out <*M*> as happens in the TM *SELF*, or count the number of states in <*M*>, or even simulate <*M*>.

Terminology for the Recursion Theorem [S-ToC-13]

♣ **Using the recursion theorem to describe *SELF***

- ***SELF* = "On any input:**
  1. Obtain, via the recursion theorem, own description <*SELF*>.
  2. Print <*SELF*>."

♣ **Using the recursion theorem to implement the "obtain own description" construct**

- **To produce the machine *SELF*, we first write the following machine *T*:**
  T = "On input <*M*, *w*>
  1. Print <*M*> and halt."
- **The TM *T* receives a description of a TM *M* and a string *w* as input, and it prints the description <*M*> of *M*.**
- **Then the recursion theorem shows how to obtain a TM *R*, which on input *w* operates like *T* on input <*R*, *w*>.**
- **Thus, *R* prints the description of *R* -- exactly what is required of *SELF*.**

Applications of the Recursion Theorem [S-ToC-13]

♣ **The proof of the undecidability of A_{TM}**

- **The recursion theorem gives us a new and simpler proof of the undecidability of A_{TM}. [$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$]**

THEOREM 6.5

A_{TM} is undecidable.

PROOF We assume that Turing machine H decides A_{TM}, for the purpose of obtaining a contradiction. We construct the following machine B.

B = "On input w:

1. Obtain, via the recursion theorem, own description $\langle B \rangle$.
2. Run H on input $\langle B, w \rangle$.
3. Do the opposite of what H says. That is, *accept* if H rejects and *reject* if H accepts."

Running B on input w does the opposite of what H declares it does. Therefore, H cannot be deciding A_{TM}. Done!

Applications of the Recursion Theorem [S-ToC-13]

♣ **Minimal TMs**

DEFINITION 6.6

If M is a Turing machine, then we say that the *length* of the description $\langle M \rangle$ of M is the number of symbols in the string describing M. Say that M is ***minimal*** if there is no Turing machine equivalent to M that has a shorter description. Let

$$MIN_{TM} = \{\langle M \rangle \mid M \text{ is a minimal TM}\}.$$

Applications of the Recursion Theorem [S-ToC-13]

♣ **Minimal TMs**

THEOREM 6.7

MIN_{TM} is not Turing-recognizable.

PROOF Assume that some TM E enumerates MIN_{TM} and obtain a contradiction. We construct the following TM C.

C = "On input w:

1. Obtain, via the recursion theorem, own description $\langle C \rangle$.
2. Run the enumerator E until a machine D appears with a longer description than that of C.
3. Simulate D on input w."

Because MIN_{TM} is infinite, E's list must contain a TM with a longer description than C's description. Therefore, step 2 of C eventually terminates with some TM D that is longer than C. Then C simulates D and so is equivalent to it. Because C is shorter than D and is equivalent to it, D cannot be minimal. But D appears on the list that E produces. Thus, we have a contradiction.

Applications of the Recursion Theorem [S-ToC-13]

♣ **The fixed-point version of the recursion theorem**

- **[D] A *fixed point* of a function is a value that is not changed by application of the function. [ex.: $f(x) = x$]**
- **In this case, we consider functions that are computable transformations of TM descriptions.**
- **We show that for any such transformation, some TM exists whose behavior is unchanged by the transformation.**

Applications of the Recursion Theorem [S-ToC-13]

♣ **The fixed-point version of the recursion theorem**

THEOREM 6.8

Let $t: \Sigma^* \longrightarrow \Sigma^*$ be a computable function. Then there is a Turing machine F for which $t(\langle F \rangle)$ describes a Turing machine equivalent to F. Here we'll assume that if a string isn't a proper Turing machine encoding, it describes a Turing machine that always rejects immediately.

In this theorem, t plays the role of the transformation, and F is the fixed point.

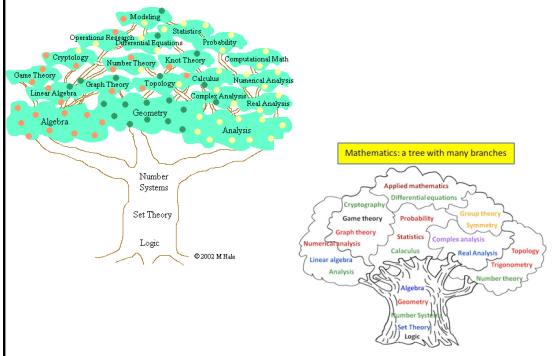
PROOF Let F be the following Turing machine.

F = "On input w:

1. Obtain, via the recursion theorem, own description $\langle F \rangle$.
2. Compute $t(\langle F \rangle)$ to obtain the description of a TM G.
3. Simulate G on w."

Clearly, $\langle F \rangle$ and $t(\langle F \rangle) = \langle G \rangle$ describe equivalent Turing machines because F simulates G.

Logic as the Fundamental Basis for all Mathematics



Decidability of Logical Theories [S-ToC-13]

♣ Classical Mathematical Logic (CML)

- CML is the branch of mathematics that investigates mathematics itself. CML addresses questions such as: What is a theorem? What is a proof? What is truth? Can an algorithm decide which statements are true? Are all true statements provable?

♣ Examples of questions

- $\forall a \forall b \forall c \forall n[(a>0 \wedge b>0 \wedge c>0 \wedge n>2) \rightarrow (a^n+b^n \neq c^n)]$ (Fermat's last theorem)
- $\forall q \exists p \forall x \forall y[p>q \wedge \{(x>1 \wedge y>1) \rightarrow (xy \neq p)\}]$ (infinitely many prime numbers exist)
- $\forall q \exists p \forall x \forall y[p>q \wedge \{(x>1 \wedge y>1) \rightarrow ((xy \neq p) \wedge (xy \neq p+2))\}]$ (twin prime conjecture)

Formal Theories: Formal Definition

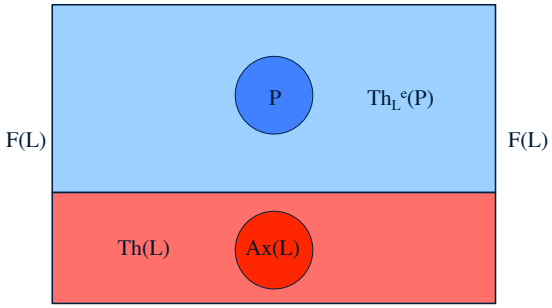
♣ L-theory with premises P

- Let L = (F(L), $\vdash_L$) be a formal logic system and $P \subseteq F$(L) be a non-empty set of *propositions / closed well-formed formulas*.
- A *formal theory* with premises P based on L, called a *L-theory with premises P* and denoted by $T_L(P)$, is defined as

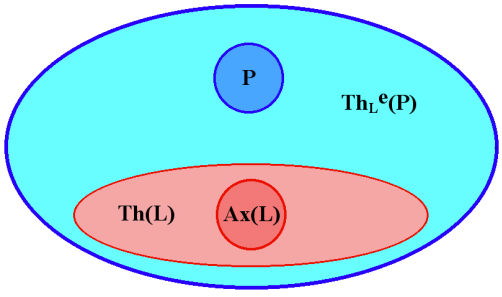
$$T_L(P) =_{df} Th(L) \cup Th_L^e(P) \text{ where } Th(L) \cap Th_L^e(P) = \varnothing$$
$$Th_L^e(P) =_{df} \{ et \mid P \vdash_L et \text{ and } et \notin Th(L) \}$$

- P: The *empirical premises / axioms*.
- Th(L): The *logical part* of the formal theory, any element of Th(L) is a *formal (logical) theorem* of that formal theory.
- $Th_L^e(P)$: The *empirical part* of the formal theory, any element of $Th_L^e(P)$ is called an *empirical theorem* of that formal theory.

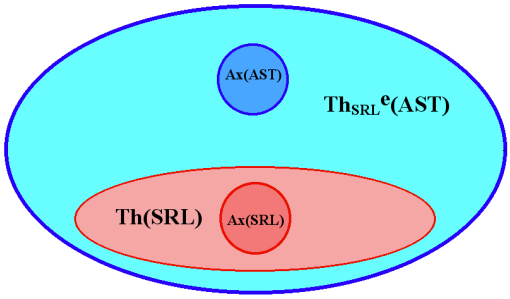
Formal Theories (L-theory with premises P, L = (F(L), $\vdash_L$))



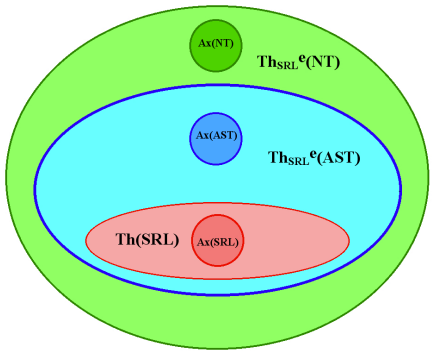
Formal Theories (L-theory with premises P, L = (F(L), $\vdash_L$))



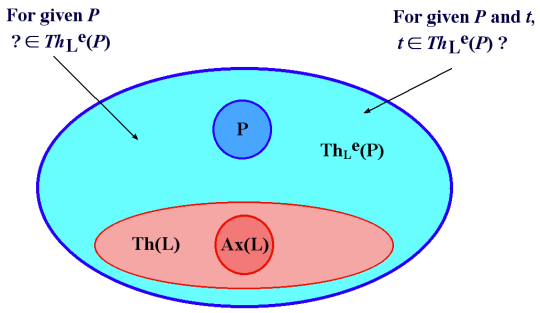
SRL-Theory of Axiomatic Set Theory



SRL-Theory of Number Theory



Reasoning Empirical Theorems vs Proving Empirical Theorems



First-Order Number Theory [Mendelson]

♣ **Number theory as the foundation for mathematics**

- Together with geometry, the theory of numbers is the most immediately intuitive of all branches of mathematics.
- It is obvious that attempts to formalize mathematics and to establish a rigorous foundation for mathematics should begin with number theory.

♣ **Peano's postulates for number theory**

- The first semi-axiomatic presentation of this subject was given by Dedekind in 1879 and, in a slightly modified form (by Peano), has come to be known as Peano's postulates.

First-Order Number Theory: Peano's Postulates [Mendelson]

♣ **Peano's postulates for number theory**

- (P1) 0 is a natural number.
- (P2) If x is a natural number, there is another natural number denoted by x' (the *successor* of x).
- (P3) $0 \neq x'$ for every natural number x.
- (P4) If $x' = y'$, then $x = y$.
- (P5) (*mathematical induction principle*) If Q is a property that may or may not hold for any given natural number, and if
  (I) 0 has the property Q, and
  (II) whenever a natural number x has the property Q, then x' has the property Q,
  then all natural numbers have the property Q.

First-Order Number Theory: The Language L_A [Mendelson]

♣ **The language of the first-order number theory: L_A (*the language of arithmetic*)**

- L_A has a single predicate symbol (letter) p^2_1. We shall abbreviate $p^2_1(t, s)$ by $t = s$, and $\neg p^2_1(t, s)$ by $t \neq s$.
- L_A has one individual constant symbol a_1. We shall use 0 as an alternative notation for a_1.
- L_A has three function symbols (letters) f^1_1, f^2_1, and f^2_2. We shall write (t') instead of $f^1_1(t)$, $(t + s)$ instead of $f^2_1(t, s)$, and $(t \times s)$ instead of $f^2_2(t, s)$. However, we shall write t', $t + s$, and $t \times s$ instead of (t'), $(t + s)$, and $(t \times s)$ whenever this will cause no confusion.

♣ **Note**

- We do not give interpretations for the three function symbols but remain them to be defined by PA axioms.

First-Order Number Theory: The Arithmetic Axioms [Mendelson]

♣ ***The arithmetic axioms of the first-order number theory (Peano Arithmetic, PA)***

- (PA1) $(\forall x_1)(\forall x_2)(\forall x_3)[x_1 = x_2 \rightarrow (x_1 = x_3 \rightarrow x_2 = x_3)]$
- (PA2) $(\forall x_1)(\forall x_2)[x_1 = x_2 \rightarrow x_1' = x_2']$
- (PA3) $(\forall x_1)[0 \neq x_1']$
- (PA4) $(\forall x_1)(\forall x_2)[x_1' = x_2' \rightarrow x_1 = x_2]$
- (PA5) $(\forall x_1)[x_1 + 0 = x_1]$
- (PA6) $(\forall x_1)(\forall x_2)[x_1 + x_2' = (x_1 + x_2)']$
- (PA7) $(\forall x_1)[x_1 \times 0 = 0]$
- (PA8) $(\forall x_1)(\forall x_2)[x_1 \times (x_2)' = (x_1 \times x_2) + x_1]$
- (PA9) $B(0) \rightarrow [(\forall x)(B(x) \rightarrow B(x')) \rightarrow (\forall x)B(x)]$ for any wff $B(x)$ of L_A. (PA9) is called *the principle of mathematical induction*.
- Notes: Axioms (PA1)-(PA8) are particular formulas, whereas (PA9) is an axiom schema providing an infinite number of axioms.

Decidability of First-Order Number Theory [Mendelson]

♣ **First-Order Number Theory: A decidable sub-theory**

- Let PA_+ [or Th(N, +)] = { (PA1)-(PA6), (PA9) }
- **Theorem:** $T_{CFOPC}(PA_+)$ **is decidable.**

♣ **First-Order Number Theory: Undecidability**

- Let $PA_{+,\times}$ [or Th(N, +, ×)] = { (PA1)-(PA9) }
- **Theorem:** $T_{CFOPC}(PA_{+,\times})$ **is undecidable.**

Decidability of First-Order Number Theory [S-ToC-13]

- **Theorem:** $T_{CFOPC}(PA_+)$ **[Th(N, +)] is decidable.**

THEOREM **6.12**

Th($\mathcal{N}$, +) is decidable.

PROOF IDEA This proof is an interesting and nontrivial application of the theory of finite automata that we presented in Chapter 1. One fact about finite automata that we use appears in Problem 1.32, (page 88) where you were asked to show that they are capable of doing addition if the input is presented in a special form. The input describes three numbers in parallel by representing one bit of each number in a single symbol from an eight-symbol alphabet. Here we use a generalization of this method to present i-tuples of numbers in parallel using an alphabet with 2^i symbols.

We give an algorithm that can determine whether its input, a sentence ϕ in the language of $\langle \mathcal{N}, + \rangle$, is true in that model. Let

$$\phi = Q_1 x_1 Q_2 x_2 \cdots Q_l x_l \, [\psi],$$

where $Q_1, \ldots, Q_l$ each represents either $\exists$ or $\forall$ and ψ is a formula without quantifiers that has variables $x_1, \ldots, x_l$. For each i from 0 to l, define formula ϕ_i as

$$\phi_i = Q_{i+1} x_{i+1} Q_{i+2} x_{i+2} \cdots Q_l x_l \, [\psi].$$

Thus $\phi_0 = \phi$ and $\phi_l = \psi$.

Formula ϕ_i has i free variables. For $a_1, \ldots, a_i \in \mathcal{N}$, write $\phi_i(a_1, \ldots, a_i)$ to be the sentence obtained by substituting the constants $a_1, \ldots, a_i$ for the variables $x_1, \ldots, x_i$ in ϕ_i.

For each i from 0 to l, the algorithm constructs a finite automaton A_i that recognizes the collection of strings representing i-tuples of numbers that make ϕ_i true. The algorithm begins by constructing A_l directly, using a generalization of the method in the solution to Problem 1.32. Then, for each i from l down to 1, it uses A_i to construct A_{i-1}. Finally, once the algorithm has A_0, it tests whether A_0 accepts the empty string. If it does, ϕ is true and the algorithm accepts.

Decidability of First-Order Number Theory [S-ToC-13]

- **Theorem:** $T_{CFOPC}(PA_{+,\times})$ **[Th(N, +, ×)] is undecidable.**

LEMMA **6.14**

Let M be a Turing machine and w a string. We can construct from M and w a formula $\phi_{M,w}$ in the language of $(\mathcal{N}, +, \times)$ that contains a single free variable x, whereby the sentence $\exists x \, \phi_{M,w}$ is true iff M accepts w.

PROOF IDEA Formula $\phi_{M,w}$ "says" that x is a (suitably encoded) accepting computation history of M on w. Of course, x actually is just a rather large integer, but it represents a computation history in a form that can be checked by using the + and × operations.

The actual construction of $\phi_{M,w}$ is too complicated to present here. It extracts individual symbols in the computation history with the + and × operations to check that the start configuration for M on w is correct, that each configuration legally follows from the one preceding it, and that the last configuration is accepting.

Decidability of First-Order Number Theory [S-ToC-13]

- **Theorem:** $T_{CFOPC}(PA_{+,\times})$ **[Th(N, +, ×)] is undecidable.**

THEOREM **6.13**

Th($\mathcal{N}$, +, ×) is undecidable.

PROOF OF THEOREM 6.13 We give a mapping reduction from A_{TM} to Th($\mathcal{N}$, +, ×). The reduction constructs the formula $\phi_{M,w}$ from the input $\langle M, w \rangle$ by using Lemma 6.14. Then it outputs the sentence $\exists x \, \phi_{M,w}$.

Gödel's Incompleteness Theorem [S-ToC-13]

♣ **Gödel's incompleteness theorem**

- **Informally, this theorem says that in any reasonable system of formalizing the notion of provability in number theory, some true statements are unprovable in the formal system.**

THEOREM **6.15**

The collection of provable statements in Th($\mathcal{N}$, +, ×) is Turing-recognizable.

PROOF The following algorithm P accepts its input ϕ if ϕ is provable. Algorithm P tests each string as a candidate for a proof π of ϕ, using the proof checker assumed in provability property 1. If it finds that any of these candidates is a proof, it accepts.

Gödel's Incompleteness Theorem [S-ToC-13]

THEOREM **6.16**

Some true statement in Th($\mathcal{N}$, +, ×) is not provable.

PROOF We give a proof by contradiction. We assume to the contrary that all true statements are provable. Using this assumption, we describe an algorithm D that decides whether statements are true, contradicting Theorem 6.13.

On input ϕ, algorithm D operates by running algorithm P given in the proof of Theorem 6.15 in parallel on inputs ϕ and $\neg\phi$. One of these two statements is true and thus by our assumption is provable. Therefore, P must halt on one of the two inputs. By provability property 2, if ϕ is provable, then ϕ is true; and if $\neg\phi$ is provable, then ϕ is false. So algorithm D can decide the truth or falsity of ϕ.

Gödel's Incompleteness Theorem [S-ToC-13]

THEOREM 6.17

The sentence $\psi_{\text{unprovable}}$, as described in the proof, is unprovable.

PROOF IDEA Construct a sentence that says "This sentence is not provable," using the recursion theorem to obtain the self-reference.

PROOF Let S be a TM that operates as follows.

S = "On any input:

1. Obtain own description $\langle S \rangle$ via the recursion theorem.
2. Construct the sentence $\psi = \neg\exists c\,[\phi_{S,0}]$, using Lemma 6.14.
3. Run algorithm P from the proof of Theorem 6.15 on input ψ.
4. If stage 3 accepts, *accept*."

Let $\psi_{\text{unprovable}}$ be the sentence ψ described in stage 2 of algorithm S. That sentence is true iff S doesn't accept 0 (the string 0 was selected arbitrarily).

If S finds a proof of $\psi_{\text{unprovable}}$, S accepts 0, and the sentence would thus be false. A false sentence cannot be provable, so this situation cannot occur. The only remaining possibility is that S fails to find a proof of $\psi_{\text{unprovable}}$ and so S doesn't accept 0. But then $\psi_{\text{unprovable}}$ is true, as we claimed.

Introduction to the Theory of Computation

- Enumerability and Diagonalization
- Finite Automata and Regular Languages
- Pushdown Automata and Context-Free Languages
- Computation: Turing Machines
- Computation: Turing-Computability (Turing-Decidability)
- Computation: Reducibility and Turing-Reducibility
- Computation: Recursive Functions
- Computation: Recursive Sets and Relations
- Equivalent Definitions of Computability
- Advanced Topics in Computability Theory
- Computational Complexity
- Time Complexity
- Space Complexity
- Intractability
- Advanced Topics in Complexity Theory