Models of Computation

- **Turing Machine**: States like Finite Automaton plus Turing tape carrying input/output and working space; head of machine working and moving on tape; updates of symbols, states and movement given by Turing table.

- **Register Machine**: Adding and Subtracting and Comparing natural numbers in registers; conditional and unconditional jumps between numbered statements.

- **Primitive recursive and \( \mu \)-recursive functions**: Functions defined from some base functions together with concatenation, primitive recursion and, in the case of \( \mu \)-recursive functions, search for places where some condition holds.
Example: Multiplication can be done naively by repeated addition.

Line 1: Function Mult($R_1$, $R_2$);
Line 2: $R_3 = 0$;
Line 3: $R_4 = 0$;
Line 4: If $R_3 = R_1$ Then Goto Line 8;
Line 5: $R_4 = R_4 + R_2$;
Line 6: $R_3 = R_3 + 1$;
Line 7: Goto Line 4;
Line 8: Return($R_4$).
Primitive Recursive: Addition, Multiplication, Subtraction, Exponentiation, Factorial, Choose-Function, Outcomes of Comparisons, Linear Functions and Polynomials.

Not Primitive Recursive: Ackermann Function:

- $f(0, y) = y + 1$;
- $f(x + 1, 0) = f(x, 1)$;
- $f(x + 1, y + 1) = f(x, f(x + 1, y))$.

Partial Recursive: Primitive recursive plus search for an input which makes function to 0.
Repetition 4

Theorem 10.23
For a partial function $f$, the following are equivalent:

- $f$ as a function from strings to strings can be computed by a Turing machine;
- $f$ as a function from natural numbers to natural numbers can be computed by a register machine;
- $f$ as a function from natural numbers to natural numbers is partial recursive.

Church’s Thesis
All reasonable models of computation over $\Sigma^*$ and $\mathbb{N}$ are equivalent and give the same notion as the partial recursive functions.
One measures the size $n$ of the input in the number of its symbols or by $\log(x) = \min\{n \in \mathbb{N} : x \leq 2^n\}$.

**Theorem 10.25**
A function $f$ is computable by a Turing machine in time $p(n)$ for some polynomial $p$ iff $f$ is computable by a register machine in time $q(n)$ for some polynomial $q$.

**Theorem 10.26**
A function $f$ is computable by a Turing machine in space $p(n)$ for some polynomial $p$ iff $f$ is computable by a register machine in such a way that all registers take at most the value $2^{q(n)}$ for some polynomial $q$.

The notions in Complexity Theory are also relatively invariant against changes of the model of computation; however, one has to interpret the word “reasonable” of Church in a stronger way than in recursion theory.
Theorem
A function is primitive recursive iff it can be computed by a register program where the only type of goto-commands which can go backwards are For-Loops, where one cannot go into or out of a For-Loop and once the For-Loop is started, its boundaries cannot be modified and the loop-variable can only be updated by the commands of the loop itself.

Remark
One can replace the Goto-commands completely by allowing only For-Loops, If-Then-Else statements and Switch-statements which are properly nested.

For full generality of Partial-Recursive functions, one would then also need While-Loops in addition to the For-Loops.
Example

Line 1: Function Factor($R_1, R_2$);
Line 2: $R_3 = R_1$;
Line 3: $R_4 = 0$;
Line 4: If $R_2 < 2$ Then Goto Line 10;
Line 5: For $R_5 = 0$ to $R_1$
Line 6: If $\text{Remainder}(R_3, R_2) > 0$ Then Goto Line 9;
Line 7: $R_3 = \text{Divide}(R_3, R_2)$;
Line 8: $R_4 = R_4 + 1$;
Line 9: Next $R_5$;
Line 10: Return($R_4$);

This function computes how often $R_2$ is a factor of $R_1$ and is primitive recursive.
Collatz Function

Not known whether primitive recursive or whether total at all.

Line 1: Function Collatz($R_1$);
Line 2: If $\text{Remainder}(R_1, 2) = 0$ Then Goto Line 6;
Line 3: If $R_1 = 1$ Then Goto Line 8;
Line 4: $R_1 = \text{Mult}(R_1, 3) + 1$;
Line 5: Goto Line 2;
Line 6: $R_1 = \text{Divide}(R_1, 2)$;
Line 7: Goto Line 2;
Line 8: Return($R_1$);

Lothar Collatz conjectured in 1937 that this function is total.
Simulating Collatz Function

Line 1: Function Collatz($R_1, R_2$);
Line 2: $LN = 2$;
Line 3: For $T = 0$ to $R_2$
Line 4: If $LN = 2$ Then Begin If Remainder($R_1, 2$) = 0
Then $LN = 6$ Else $LN = 3$; Goto Line 10 End;
Line 5: If $LN = 3$ Then Begin If $R_1 = 1$ Then $LN = 8$
Else $LN = 4$; Goto Line 10 End;
Line 6: If $LN = 4$ Then Begin $R_1 = \text{Mult}(R_1, 3) + 1$;
$LN = 5$; Goto Line 10 End;
Line 7: If $LN = 5$ Then Begin $LN = 2$; Goto Line 10 End;
Line 8: If $LN = 6$ Then Begin $R_1 = \text{Divide}(R_1, 2)$;
$LN = 7$; Goto Line 10 End;
Line 9: If $LN = 7$ Then Begin $LN = 2$; Goto Line 10 End;
Line 10: Next $T$;
Line 11: If $LN = 8$ Then Return($R_1 + 1$) Else Return(0).
Exercise 11.1

Write a program for a primitive recursive function which simulate the following function with input $R_1$ for $R_2$ steps.

Line 1: Function Expo($R_1$);
Line 2: $R_3 = 1$;
Line 3: If $R_1 = 0$ Then Goto Line 7;
Line 4: $R_3 = R_3 + R_3$;
Line 5: $R_1 = R_1 - 1$;
Line 6: Goto Line 3;
Line 7: Return($R_3$).
Exercise 11.2

Write a program for a primitive recursive function which simulate the following function with input $R_1$ for $R_2$ steps.

Line 1: Function Repeatadd($R_1$);
Line 2: $R_3 = 3$;
Line 3: If $R_1 = 0$ Then Goto Line 7;
Line 4: $R_3 = R_3 + R_3 + R_3 + 3$;
Line 5: $R_1 = R_1 - 1$;
Line 6: Goto Line 3;
Line 7: Return($R_3$).
Bounded Simulation

Theorem 11.3
For every partial-recursive function $f$ there is a primitive recursive function $g$ and a register machine $M$ such that for all $t$,

If $f(x_1, \ldots, x_n)$ is computed by $M$ within $t$ steps
Then $g(x_1, \ldots, x_n, t) = f(x_1, \ldots, x_n) + 1$
Else $g(x_1, \ldots, x_n, t) = 0$.

In short words, $g$ simulates the program $M$ of $f$ for $t$ steps and if an output $y$ comes then $g$ outputs $y + 1$ else $g$ outputs $0$. 

Theorem 11.4
The following notions are equivalent for a set \( A \subseteq \mathbb{N} \):
(a) \( A \) is the range of a partial recursive function;
(b) \( A \) is empty or \( A \) is the range of a total recursive function;
(c) \( A \) is empty or \( A \) is the range of a primitive recursive function;
(d) \( A \) is the set of inputs on which some register machine terminates;
(e) \( A \) is the domain of a partial recursive function;
(f) There is a two-place recursive function \( g \) such that
\[
A = \{ x : \exists y \ [ g(x, y) > 1] \}.
\]

Definition 11.5
The set \( A \) is recursively enumerable iff it satisfies any of the above equivalent properties.
(a) to (c) and (c) to (b)

If $A$ is empty then (c) holds; if $A$ is not empty then there is an element $a \in A$ which is now taken as a constant. For the partial function $f$ whose range $A$ is, there is, by Theorem 11.3, a primitive function $g$ such that either $g(x, t) = 0$ or $g(x, t) = f(x) + 1$ and whenever $f(x)$ takes a value there is also a $t$ with $g(x, t) = f(x) + 1$. Now one defines a new function $h$ which is also primitive recursive such that if $g(x, t) = 0$ then $h(x, t) = a$ else $h(x, t) = g(x, t) - 1$. The range of $h$ is $A$.

(c) $\Rightarrow$ (b): This follows by definition as every primitive recursive function is also recursive.
(b) ⇒ (d): Given a function \( h \) whose range is \( A \), one can make a register machine which simulates \( h \) and searches over all possible inputs and checks whether \( h \) on these inputs is \( x \). If such inputs are found then the search terminates else the register machine runs forever. Thus \( x \in A \) iff the register machine program following this behaviour terminates after some time.

(d) ⇒ (e): The domain of a register machine is the set of inputs on which it halts and outputs a return value. Thus this implication is satisfied trivially by taking the function for (e) to be exactly the function computed from the register program for (d).
(e) to (f) and (f) to (a)

(e) ⇒ (f): Given a register program \( f \) whose domain \( A \) is according to (e), one takes the function \( g \) as defined by Theorem 11.3 and this function indeed satisfies that \( f(x) \) is defined iff there is a \( t \) such that \( g(x, t) > 0 \).

(f) ⇒ (a): Given the function \( g \) as defined in (f), one defines that if there is a \( t \) with \( g(x, t) > 0 \) then \( f(x) = x \) else \( f(x) \) is undefined. The latter comes by infinite search for a \( t \) which is not found. Thus the partial recursive function \( f \) has range \( A \).
Decidable and Undecidable Problems

A set $L$ is called **decidable** or **recursive** iff there is a recursive function $f$ such that, for all $x$, if $x \in L$ then $f(x) = 1$ else $f(x) = 0$. One says that the function $f$ **decides** the membership in $L$.

A set $L$ is called **undecidable** or **nonrecursive** iff there is no such recursive function $f$ deciding the membership in $L$.

**Observation**
Every recursive set is recursively enumerable.
The Halting Problem

Definition [Turing 1936]
Let $e, x \mapsto \varphi_e(x)$ be a universal partial recursive function covering all one-variable partial recursive functions. Then the set $\{(e, x) : \varphi_e(x) \text{ is defined}\}$ is called the general halting problem and $K = \{e : \varphi_e(e) \text{ is defined}\}$ is called the diagonal halting problem.

The name stems from the fact that $\varphi_e(x)$ is defined iff the $e$-th register machine with input $x$ halts and produces some output.

Theorem [Turing 1936]
Both the diagonal halting problem and the general halting problem are recursively enumerable and undecidable.
Proof

Let F be a function which simulates $\varphi_e(x)$ and assume that there is a function $\text{Halt}$ which can check whether $\varphi_e(e)$ halts. If so, then $\text{Halt}(e) = 1$ else $\text{Halt}(e) = 0$. Now consider this program.

Line 1: Function Diagonalise($R_1$);
Line 2: $R_2 = 0$;
Line 3: If $\text{Halt}(R_1) = 0$ Then Goto Line 5;
Line 4: $R_2 = F(R_1, R_1) + 1$;
Line 5: Return($R_2$).
Function Diagonalise

The function Diagonalise has only one input.

If \( \varphi_e(e) \) is undefined then \( \text{Halt}(e) = 0 \) and \( \text{Diagonalise}(e) = 0 \).

If \( \varphi_e(e) \) is defined then \( \text{Halt}(e) = 1 \) and \( F(e, e) = \varphi_e(e) \) will be computed in Line 4 and the output will be \( \varphi_e(e) + 1 \).

Thus \( \text{Diagonalise}(e) \) differs from \( \varphi_e(e) \) for all \( e \) and is not among \( \varphi_0, \varphi_1, \ldots \); as all partial-recursive functions with one input are in this list, \text{Diagonalise} cannot be recursive and therefore \text{Halt} also cannot be recursive.

The halting problem equals \( \{(e, x) : F(e, x) \text{ halts} \} \). Thus it is the domain of a partial recursive function and recursively enumerable. Similarly, \( K = \{e : F(e, e) \text{ halts} \} \) is the domain of a partial-recursive function and recursively enumerable.
R.E. and Recursive

Theorem 11.9
A set $L$ is recursive iff both $L$ and $\mathbb{N} - L$ are recursively enumerable.

Exercise 11.10
Prove this connection.

Exercises 11.11 – 11.13
Prove that the following variants of the halting problem are undecidable:
11.11: $\{e : \varphi_e(2e + 5) \text{ is defined}\}$;
11.12: $\{e : \varphi_e(e^2 + 1) \text{ is defined}\}$;
11.13: $\{e : \varphi_e(e/2) \text{ is defined}\}$, where $1/2$ is rounded to 0 and $3/2$ to 1 and so on.
Further Homeworks

Show that the following sets are recursively enumerable by proving that a register machine halts exactly on the members of the set:
Exercise 11.14: \( \{ x \in \mathbb{N} : x \text{ is a square} \} \).
Exercise 11.15: \( \{ x \in \mathbb{N} : x \text{ is prime} \} \).

Exercise 11.16
Prove that the set \( \{ e : \varphi_e(e/2) \text{ is defined} \} \) is recursively enumerable by proving that it is the range of a primitive recursive function. Here \( e/2 \) is the downrounded value of \( e \) divided by 2, so \( 1/2 \) is 0 and \( 3/2 \) is 1.

Second Half of Lecture: Midterm Test