Theory of Computation

Course note based on *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*, 2nd edition, authored by Martin Davis, Ron Sigal, and Elaine J. Weyuker.

course note prepared by

Tyng–Ruey Chuang

Institute of Information Science, Academia Sinica

Department of Information Management, National Taiwan University

Week 5, Spring 2008

About This Course Note

- It is prepared for the course *Theory of Computation* taught at the National Taiwan University in Spring 2008.
- It follows very closely the book *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*, 2nd edition, by Martin Davis, Ron Sigal, and Elaine J. Weyuker. Morgan Kaufmann Publishers. ISBN: 0-12-206382-1.
- It is available from Tyng-Ruey Chuang's web site:

  `http://www.iis.sinica.edu.tw/~trc/`

  and released under a Creative Commons "Attribution-ShareAlike 2.5 Taiwan" license:

  `http://creativecommons.org/licenses/by-sa/2.5/tw/`

Compute with Numbers of Programs

- Programs taking programs as input: Compilers, interpreters, evaluators, Web browsers, ...

Compute with Numbers of Programs

- Programs taking programs as input: Compilers, interpreters, evaluators, Web browsers, ...
- Can we write a program in language $\mathscr{S}$ to accept the number of another program $\mathscr{P}$, as well as the input x to $\mathscr{P}$, then compute $\Psi^{(1)}_{\mathscr{P}}(x)$ as output?

Compute with Numbers of Programs

- Programs taking programs as input: Compilers, interpreters, evaluators, Web browsers, ...
- Can we write a program in language $\mathscr{S}$ to accept the number of another program $\mathscr{P}$, as well as the input x to $\mathscr{P}$, then compute $\Psi_{\mathscr{P}}^{(1)}(x)$ as output?
- Yes, we can! The program above is called an universal program.

Universality

For each $n > 0$, we define

$$\Phi^{(n)}(x_1, \ldots, x_n, y) = \Psi^{(n)}_{\mathscr{P}}(x_1, \ldots, x_n), \quad \text{where } \#(\mathscr{P}) = y.$$

Universality

For each $n > 0$, we define

$$\Phi^{(n)}(x_1, \ldots, x_n, y) = \Psi^{(n)}_{\mathscr{P}}(x_1, \ldots, x_n), \quad \text{where } \#(\mathscr{P}) = y.$$

Theorem 3.1. For each $n > 0$, the function $\Phi^{(n)}(x_1, \ldots, x_n, y)$ is partially computable. □

Universality

For each $n > 0$, we define

$$\Phi^{(n)}(x_1, \ldots, x_n, y) = \Psi^{(n)}_{\mathscr{P}}(x_1, \ldots, x_n), \quad \text{where } \#(\mathscr{P}) = y.$$

Theorem 3.1. For each $n > 0$, the function $\Phi^{(n)}(x_1, \ldots, x_n, y)$ is partially computable. □

We shall prove this theorem by showing how to construct, for each $n > 0$, a program $\mathscr{U}_n$ which computes $\Phi^{(n)}$. That is,

$$\Psi^{(n+1)}_{\mathscr{U}_n}(x_1, \ldots, x_n, x_{n+1}) = \Phi^{(n)}(x_1, \ldots, x_n, x_{n+1}).$$

The programs $\mathscr{U}_n$ are called universal.

"Computer Organization" of $\mathscr{U}_n$

- Program $\mathscr{U}_n$ accepts $n+1$ input variables of which X_{n+1} is a number of a program $\mathscr{P}$, and $X_1, \ldots, X_n$ are provided to $\mathscr{P}$ as input variables.
- All variables used by $\mathscr{P}$ are arranged in the following order

$$Y, X_1, Z_1, X_2, Z_2, \ldots$$

  and their state is coded by the Gödel number $[y, x_1, z_1, x_2, z_2, \ldots]$.
- Let variable S in program $\mathscr{U}_n$ store the current state of program $\mathscr{P}$ coded in the above manner.
- Let variable K in program $\mathscr{U}_n$ store the number such that the Kth instruction of program $\mathscr{P}$ is about to be executed.
- Let variable Z in program $\mathscr{U}_n$ store the instruction sequence of program $\mathscr{P}$ coded as a Gödel number.

Setting Up

As program $\mathscr{U}_n$ computes $\Phi^{(n)}(X_1, \ldots, X_n, X_{n+1})$, we begin $\mathscr{U}_n$ by setting up the initial environment for program (number) X_{n+1} to execute:

$$Z \leftarrow X_{n+1} + 1$$
$$S \leftarrow \prod_{i=1}^{n} (p_{2i})^{X_i}$$
$$K \leftarrow 1$$

Setting Up

As program $\mathscr{U}_n$ computes $\Phi^{(n)}(X_1, \ldots, X_n, X_{n+1})$, we begin $\mathscr{U}_n$ by setting up the initial environment for program (number) X_{n+1} to execute:

$$Z \leftarrow X_{n+1} + 1$$
$$S \leftarrow \prod_{i=1}^{n} (p_{2i})^{X_i}$$
$$K \leftarrow 1$$

- If $X_{n+1} = \#(\mathscr{P})$, where $\mathscr{P}$ consists of instructions $I_1, \ldots, I_m$, then Z gets the value $[\#(I_1), \ldots, \#(I_m)]$.

Setting Up

As program $\mathscr{U}_n$ computes $\Phi^{(n)}(X_1, \ldots, X_n, X_{n+1})$, we begin $\mathscr{U}_n$ by setting up the initial environment for program (number) X_{n+1} to execute:

$$Z \leftarrow X_{n+1} + 1$$
$$S \leftarrow \prod_{i=1}^{n} (p_{2i})^{X_i}$$
$$K \leftarrow 1$$

- If $X_{n+1} = \#(\mathscr{P})$, where $\mathscr{P}$ consists of instructions $I_1, \ldots, I_m$, then Z gets the value $[\#(I_1), \ldots, \#(I_m)]$.
- S is initialized as $[0, X_1, 0, X_2, \ldots, 0, X_n]$ which gives the first n input variables their appropriate values and gives all other variables the value 0.

Setting Up

As program $\mathscr{U}_n$ computes $\Phi^{(n)}(X_1, \ldots, X_n, X_{n+1})$, we begin $\mathscr{U}_n$ by setting up the initial environment for program (number) X_{n+1} to execute:

$$Z \leftarrow X_{n+1} + 1$$
$$S \leftarrow \prod_{i=1}^{n} (p_{2i})^{X_i}$$
$$K \leftarrow 1$$

- If $X_{n+1} = \#(\mathscr{P})$, where $\mathscr{P}$ consists of instructions $I_1, \ldots, I_m$, then Z gets the value $[\#(I_1), \ldots, \#(I_m)]$.
- S is initialized as $[0, X_1, 0, X_2, \ldots, 0, X_n]$ which gives the first n input variables their appropriate values and gives all other variables the value 0.
- K, the instruction counter, is given the initial value 1.

Decoding Instruction

We first see if the execution of program $\mathscr{P}$ shall halt. If not, we fetch the Kth instruction and decode the instruction.

$[C]$ IF $K = Lt(Z) + 1 \ \vee \ K = 0$ GOTO F
$U \leftarrow r((Z)_k)$
$P \leftarrow p_{r(U)+1}$

Decoding Instruction

We first see if the execution of program $\mathscr{P}$ shall halt. If not, we fetch the Kth instruction and decode the instruction.

$[C]$ IF $K = Lt(Z) + 1 \ \vee \ K = 0$ GOTO F
$U \leftarrow r((Z)_k)$
$P \leftarrow p_{r(U)+1}$

- If the computation has ended, GOTO F, where the proper value will be output. (The case for $K = 0$ will be explained later.)

Decoding Instruction

We first see if the execution of program $\mathscr{P}$ shall halt. If not, we fetch the Kth instruction and decode the instruction.

$[C]$ IF $K = Lt(Z) + 1 \ \vee \ K = 0$ GOTO F
$U \leftarrow r((Z)_k)$
$P \leftarrow p_{r(U)+1}$

- If the computation has ended, GOTO F, where the proper value will be output. (The case for $K = 0$ will be explained later.)
- $(Z)_k = \langle a, \langle b, c \rangle \rangle$ is the number of the Kth instruction. Thus $U = \langle b, c \rangle$ is the code of the statement to be executed.

Decoding Instruction

We first see if the execution of program $\mathscr{P}$ shall halt. If not, we fetch the Kth instruction and decode the instruction.

$[C]$ IF $K = Lt(Z) + 1 \ \vee \ K = 0$ GOTO F
$U \leftarrow r((Z)_k)$
$P \leftarrow p_{r(U)+1}$

- If the computation has ended, GOTO F, where the proper value will be output. (The case for $K = 0$ will be explained later.)
- $(Z)_k = \langle a, \langle b, c \rangle \rangle$ is the number of the Kth instruction. Thus $U = \langle b, c \rangle$ is the code of the statement to be executed.
- The variable mentioned in the statement is the $(r(U) + 1)$th in our list, and its current value is stored as the exponent to which P divides S.

Instruction Execution

IF $I(U) = 0$ GOTO N
IF $I(U) = 1$ GOTO A
IF $\sim (P|S)$ GOTO N
IF $I(U) = 2$ GOTO M

Instruction Execution

IF $I(U) = 0$ GOTO N
IF $I(U) = 1$ GOTO A
IF $\sim (P|S)$ GOTO N
IF $I(U) = 2$ GOTO M

- If $I(U) = 0$, the instruction is a dummy $\boxed{V \leftarrow V}$ and the computation does nothing. Hence, it goes to N (for *Nothing*).

Instruction Execution

IF $I(U) = 0$ GOTO N
IF $I(U) = 1$ GOTO A
IF $\sim (P|S)$ GOTO N
IF $I(U) = 2$ GOTO M

- If $I(U) = 0$, the instruction is a dummy $\boxed{V \leftarrow V}$ and the computation does nothing. Hence, it goes to N (for *Nothing*).
- If $I(U) = 1$, the instruction is $\boxed{V \leftarrow V + 1}$. The computation goes to A (for *Add*) to add 1 to the exponent on P in the prime power factorization of S.

Instruction Execution

IF $I(U) = 0$ GOTO N
IF $I(U) = 1$ GOTO A
IF $\sim (P|S)$ GOTO N
IF $I(U) = 2$ GOTO M

- If $I(U) = 0$, the instruction is a dummy $\boxed{V \leftarrow V}$ and the computation does nothing. Hence, it goes to N (for *Nothing*).
- If $I(U) = 1$, the instruction is $\boxed{V \leftarrow V + 1}$. The computation goes to A (for *Add*) to add 1 to the exponent on P in the prime power factorization of S.
- If $I(U) \neq 0, 1$, the instruction is either $\boxed{V \leftarrow V - 1}$, or $\boxed{\text{IF } V \neq 0 \text{ GOTO } L}$. In both cases, if $V = 0$, the computation does nothing so goes to N. This happens when P is not a divisor of S.

Instruction Execution

IF $I(U) = 0$ GOTO N
IF $I(U) = 1$ GOTO A
IF $\sim (P|S)$ GOTO N
IF $I(U) = 2$ GOTO M

- If $I(U) = 0$, the instruction is a dummy $\boxed{V \leftarrow V}$ and the computation does nothing. Hence, it goes to N (for *Nothing*).
- If $I(U) = 1$, the instruction is $\boxed{V \leftarrow V + 1}$. The computation goes to A (for *Add*) to add 1 to the exponent on P in the prime power factorization of S.
- If $I(U) \neq 0, 1$, the instruction is either $\boxed{V \leftarrow V - 1}$, or $\boxed{\text{IF } V \neq 0 \text{ GOTO } L}$. In both cases, if $V = 0$, the computation does nothing so goes to N. This happens when P is not a divisor of S.
- If $P|S$ and $I(U) = 2$, the computation goes to M (for *Minus*).

Branching

$K \leftarrow \min_{i \leq Lt(Z)}[I((Z)_i) + 2 = I(U)]$
GOTO C

Branching

$K \leftarrow \min_{i \leq Lt(Z)}[I((Z)_i) + 2 = I(U)]$
GOTO C

- If $I(U) > 2$ and $P|S$, the current instruction is of the form $\boxed{\text{IF } V \neq 0 \text{ GOTO } L}$ where V has a nonzero value and L is the label whose position in our label list is $I(U) - 2$.

Branching

$K \leftarrow \min_{i \leq Lt(Z)}[I((Z)_i) + 2 = I(U)]$
GOTO C

- If $I(U) > 2$ and $P|S$, the current instruction is of the form $\boxed{\text{IF } V \neq 0 \text{ GOTO } L}$ where V has a nonzero value and L is the label whose position in our label list is $I(U) - 2$.
- The next instruction should be the first with this label.

Branching

$K \leftarrow \min_{i \leq Lt(Z)}[I((Z)_i) + 2 = I(U)]$
GOTO C

- If $I(U) > 2$ and $P|S$, the current instruction is of the form $\boxed{\text{IF } V \neq 0 \text{ GOTO } L}$ where V has a nonzero value and L is the label whose position in our label list is $I(U) - 2$.
- The next instruction should be the first with this label.
- That is, K should get as its value the least i for which $I((Z)_i) = I(U) - 2$. If there is no instruction with the appropriate label, K gets the 0, which will lead to termination the next time through the main loop.

Branching

$K \leftarrow \min_{i \leq Lt(Z)}[I((Z)_i) + 2 = I(U)]$
GOTO C

- If $I(U) > 2$ and $P|S$, the current instruction is of the form IF $V \neq 0$ GOTO L where V has a nonzero value and L is the label whose position in our label list is $I(U) - 2$.
- The next instruction should be the first with this label.
- That is, K should get as its value the least i for which $I((Z)_i) = I(U) - 2$. If there is no instruction with the appropriate label, K gets the 0, which will lead to termination the next time through the main loop.
- Once the instruction counter K is adjusted, the execution enters the main loop by GOTO C.

Subtraction and Addition

$[M]$ $S \leftarrow \lfloor S/P \rfloor$
GOTO N
$[A]$ $S \leftarrow S \cdot P$
$[N]$ $K \leftarrow K + 1$
GOTO C

Subtraction and Addition

$[M]$ $S \leftarrow \lfloor S/P \rfloor$
GOTO N
$[A]$ $S \leftarrow S \cdot P$
$[N]$ $K \leftarrow K + 1$
GOTO C

- 1 is subtracted from the variable by dividing S by P.

Subtraction and Addition

$[M]$ $S \leftarrow \lfloor S/P \rfloor$
GOTO N
$[A]$ $S \leftarrow S \cdot P$
$[N]$ $K \leftarrow K + 1$
GOTO C

- 1 is subtracted from the variable by dividing S by P.
- 1 is added to the variable by multiplying S by P.

Subtraction and Addition

$[M]$ $S \leftarrow \lfloor S/P \rfloor$
GOTO N
$[A]$ $S \leftarrow S \cdot P$
$[N]$ $K \leftarrow K + 1$
GOTO C

- 1 is subtracted from the variable by dividing S by P.
- 1 is added to the variable by multiplying S by P.
- The instruction counter is increased by 1 and the computation returns to the main loop to fetch the next instruction.

Finalizing

$[F] \quad Y \leftarrow (S)_1$

Finalizing

$[F] \quad Y \leftarrow (S)_1$

- One termination, the value of Y for the program being simulated is stored at the exponent on p_1 in S.

$\mathscr{U}_n$, Finally

$Z \leftarrow X_{n+1} + 1$
$S \leftarrow \prod_{i=1}^{n} (p_{2i})^{X_i}$
$K \leftarrow 1$
$[C]$ IF $K = Lt(Z) + 1 \ \vee \ K = 0$ GOTO F
$U \leftarrow r((Z)_k)$
$P \leftarrow p_{r(U)+1}$
IF $I(U) = 0$ GOTO N
IF $I(U) = 1$ GOTO A
IF $\sim (P|S)$ GOTO N
IF $I(U) = 2$ GOTO M
$K \leftarrow \min_{i \leq Lt(Z)}[I((Z)_i) + 2 = I(U)]$
GOTO C
$[M]$ $S \leftarrow \lfloor S/P \rfloor$
GOTO N
$[A]$ $S \leftarrow S \cdot P$
$[N]$ $K \leftarrow K + 1$
GOTO C
$[F]$ $Y \leftarrow (S)_1$

Notations

For each $n > 0$, the sequence

$$\Phi^{(n)}(x_1, \ldots, x_n, 0), \Phi^{(n)}(x_1, \ldots, x_n, 1), \ldots$$

enumerates all partially computable functions of n variables. When we want to emphasize this aspect we write

$$\Phi_y^{(n)}(x_1, \ldots, x_n) = \Phi^{(n)}(x_1, \ldots, x_n, y)$$

It is often convenient to omit the superscript when $n = 1$, writing

$$\Phi_y(x) = \Phi(x, y) = \Phi^{(1)}(x, y).$$

The Stepping Function STP

A simple modification of the program $\mathscr{U}_n$ would enable us to prove that following predicate is computable:

$$\begin{aligned} \mathrm{STP}(x_1, \ldots, x_n, y, t) \quad &\Leftrightarrow \quad \text{Program number } y \text{ halts after} \\ &\quad\ \ t \text{ or fewer steps on inputs } x_1, \ldots, x_n \\ &\Leftrightarrow \quad \text{There is a computation of program } y \\ &\quad\ \ \text{of length} \leq t+1, \text{ beginning with} \\ &\quad\ \ \text{inputs } x_1, \ldots, x_n \end{aligned}$$

We simply need to add a counter to determine when we have simulated t steps.

However, we can prove a stronger result.

Function STP is Primitive Recursive

Theorem 3.2. For each $n > 0$, the predicate $\mathrm{STP}^{(n)}(x_1, \ldots, x_n, y, t)$ is primitive recursive. □

The idea is to provide numeric versions of the notations of snapshot and successor of snapshot, and to show that the necessary functions are primitive recursive.

We first define the following functions for extracting the components of the ith instruction of program number y:

$$\begin{aligned} \mathrm{LABEL}(i, y) &= l((y+1)_i) \\ \mathrm{VAR}(i, y) &= r(r((y+1)_i))) + 1 \\ \mathrm{INSTR}(i, y) &= l(r((y+1)_i)) \\ \mathrm{LABEL}'(i, y) &= l(r((y+1)_i)) \dot{-} 2 \end{aligned}$$

Function STP is Primitive Recursive, Continued

Next we define some predicates that indicate, for program y and the snapshot represented by $x = \langle i, s \rangle$, where i is the number of the instruction to be executed and s the current state (i.e., variable S in $\mathscr{U}_n$), what kind of action is to be performed next.

$$
\begin{aligned}
\mathrm{SKIP}(x,y) &\Leftrightarrow [\mathrm{INSTR}(I(x),y) = 0 \;\&\; I(x) \le Lt(y+1)] \\
&\quad \vee [\mathrm{INSTR}(I(x),y) \ge 2 \;\&\; \sim (p_{\mathrm{VAR}(I(x),y)} | r(x))] \\
\mathrm{INCR}(x,y) &\Leftrightarrow \mathrm{INSTR}(I(x),y) = 1 \\
\mathrm{DECR}(x,y) &\Leftrightarrow \mathrm{INSTR}(I(x),y) = 2 \;\&\; p_{\mathrm{VAR}(I(x),y)} | r(x) \\
\mathrm{BRANCH}(x,y) &\Leftrightarrow \mathrm{INSTR}(I(x),y) > 2 \;\&\; p_{\mathrm{VAR}(I(x),y)} | r(x) \\
&\quad \;\&\; (\exists i)_{\le Lt(y+1)} \mathrm{LABEL}(i,y) = \mathrm{LABEL}'(I(x),y)
\end{aligned}
$$

Function STP is Primitive Recursive, Continued

Now we can define $\mathrm{SUCC}(x, y)$, which for program number y, computes the successor to the snapshot represented by x.

$$\mathrm{SUCC}(x,y) = \begin{cases} \langle l(x)+1,\ r(x)\rangle & \text{if } \mathrm{SKIP}(x,y) \\ \langle l(x)+1,\ r(x)\cdot p_{\mathrm{VAR}(l(x),y)}\rangle & \text{if } \mathrm{INCR}(x,y) \\ \langle l(x)+1,\ r(x)/p_{\mathrm{VAR}(l(x),y)}\rangle & \text{if } \mathrm{DECR}(x,y) \\ \min_{i\le Lt(y+1)}[\mathrm{LABEL}(i,y)=\mathrm{LABEL}'(l(x),y)],\ r(x)\rangle & \text{if } \mathrm{BRANCH}(x,y) \\ \langle Lt(y+1)+1,\ r(x)\rangle & \text{otherwise.} \end{cases}$$

Function STP is Primitive Recursive, Continued

We also need

$$\text{INIT}^{(n)}(x_1, \ldots, x_n) = \langle 1, \prod_{i=1}^{n} (p_{2i})^{x_i} \rangle$$

which gives the representation of the initial snapshot for input $x_1, \ldots, x_n$, and

$$\text{TERM}(x, y) \Leftrightarrow l(x) > Lt(y + 1)$$

which tests whether x represents a terminal snapshot for program y.

Function STP is Primitive Recursive, Continued

Putting these together we can define a primitive recursive function that gives the numbers of the successive snapshots produced by a given program:

$$\begin{aligned} \mathrm{SNAP}^{(n)}(x_1, \ldots, x_n, y, 0) &= \mathrm{INIT}^{(n)}(X_1, \ldots, x_n) \\ \mathrm{SNAP}^{(n)}(x_1, \ldots, x_n, y, i+1) &= \mathrm{SUCC}(\mathrm{SNAP}^{(n)}(x_1, \ldots, x_n, y, i), y) \end{aligned}$$

Thus,

$$\mathrm{STP}^{(n)}(x_1, \ldots, x_n, y, t) \Leftrightarrow \mathrm{TERM}(\mathrm{SNAP}^{(n)}(x_1, \ldots, x_n, y, t), y)$$

It is clear that $\mathrm{STP}^{(n)}(x_1, \ldots, x_n, y, t)$ is primitive recursive. □

A Normal Form for Partial Computable Functions

Theorem 3.3. Let $f(x_1, \ldots, x_n)$ be a partially computable function. Then there is a primitive recursive predicate $R(x_1, \ldots, x_n, y)$ such that

$$f(x_1, \ldots, x_n) = l(\min_z R(x_1, \ldots, x_n, z))$$

□

Proof. Let y_0 be the number of a program that computes $f(x_1, \ldots, x_n)$. Let $R(x_1, \ldots, x_n, z)$ be the following predicate

$$\begin{aligned} & \mathrm{STP}^{(n)}(x_1, \ldots, x_n, y_0, r(z)) \\ \& \quad & (r(\mathrm{SNAP}^{(n)}(x_1, \ldots, x_n, y_0, r(z))))_1 = l(z) \end{aligned}$$

A Normal Form for Partial Computable Functions

Theorem 3.3. Let $f(x_1, \ldots, x_n)$ be a partially computable function. Then there is a primitive recursive predicate $R(x_1, \ldots, x_n, y)$ such that

$$f(x_1, \ldots, x_n) = l(\min_z R(x_1, \ldots, x_n, z))$$

□

Proof. Let y_0 be the number of a program that computes $f(x_1, \ldots, x_n)$. Let $R(x_1, \ldots, x_n, z)$ be the following predicate

$$\begin{aligned} & \mathrm{STP}^{(n)}(x_1, \ldots, x_n, y_0, r(z)) \\ \& \quad & (r(\mathrm{SNAP}^{(n)}(x_1, \ldots, x_n, y_0, r(z))))_1 = l(z) \end{aligned}$$

If the above predicate is defined, then for any such z, the computation by program y_0 terminates in $r(z)$ or few steps, and $l(z)$ is the value held in the output variable Y.

A Normal Form for Partial Computable Functions

Theorem 3.3. Let $f(x_1, \ldots, x_n)$ be a partially computable function. Then there is a primitive recursive predicate $R(x_1, \ldots, x_n, y)$ such that

$$f(x_1, \ldots, x_n) = l(\min_z R(x_1, \ldots, x_n, z))$$

□

Proof. Let y_0 be the number of a program that computes $f(x_1, \ldots, x_n)$. Let $R(x_1, \ldots, x_n, z)$ be the following predicate

$$\begin{aligned} & \mathrm{STP}^{(n)}(x_1, \ldots, x_n, y_0, r(z)) \\ \& \quad & (r(\mathrm{SNAP}^{(n)}(x_1, \ldots, x_n, y_0, r(z))))_1 = l(z) \end{aligned}$$

If the above predicate is defined, then for any such z, the computation by program y_0 terminates in $r(z)$ or few steps, and $l(z)$ is the value held in the output variable Y. If on the other hand, the predicate is undefined, then $\mathrm{STP}^{(n)}(x_1, \ldots, x_n, y_0, t)$ must be false for all t. That is, $f(x_1, \ldots, x_n)\uparrow$. □

A Characterization of Partially Computable Functions

Theorem 3.4. A function is partially computable if and only if it can be obtained from the initial functions by a finite number of applications of composition, recursion, and minimalization. □

A Characterization of Partially Computable Functions

Theorem 3.4. A function is partially computable if and only if it can be obtained from the initial functions by a finite number of applications of composition, recursion, and minimalization. □

Proof. ($\Leftarrow$) That every function which can be so obtained is partially computable is a consequence of Theorems 1.1, 2.1, 2,2, 3.1, and 7.2 in Chapter 3. That is, there is a program in $\mathscr{S}$ for the function so obtained.

A Characterization of Partially Computable Functions

Theorem 3.4. A function is partially computable if and only if it can be obtained from the initial functions by a finite number of applications of composition, recursion, and minimalization. □

Proof. ($\Leftarrow$) That every function which can be so obtained is partially computable is a consequence of Theorems 1.1, 2.1, 2,2, 3.1, and 7.2 in Chapter 3. That is, there is a program in $\mathscr{S}$ for the function so obtained.
($\Rightarrow$) Given a partially computable function — a program in language $\mathscr{S}$ — Theorem 3.3 in this Chapter show how to express this function in the form

$$I(\min_z R(x_1, \ldots, x_n, z))$$

where R is a primitive recursive predicate. Finally, R is used in a minimalization and then composed with the primitive recursive function I. □

A Characterization of Computable Functions

When $\min_z R(x_1, \ldots, x_n, z)$ is a total function, we say that we are applying the operation of *proper* minimalization to R. Now, if

$$I(\min_z R(x_1, \ldots, x_n, z))$$

is total, then $R(x_1, \ldots, x_n, z)$ must be total too. Hence we have:

Theorem 3.5. A function is computable if and only if it can be obtained from the initial functions by a finite number of applications of composition, recursion, and *proper* minimalization. □