# 10. ‘loop’ Programs

## 10.1 Definition

Up to now our development of computability theory was done in terms of the $\mathcal{G}$ programming language. We have asserted (in §7.1) the equivalence of this notion with many other notions of computability, and proved (in Section 9) its equivalence to $\mu$-primitive recursiveness. In this section, and the next, we turn to ***other simple programming languages***, and investigate whether the corresponding notions of computability are equivalent to $\mathcal{G}$-computability or not.

First we consider the programming language $\mathcal{L}$ (for “loop”), with the *instructions*

$$
\begin{array}{l}
V \leftarrow 0 \\
V \leftarrow W \\
V{+}{+} \\
\left\{\begin{array}{l} \mathsf{loop}\ V \\ \quad\vdots \\ \mathsf{end} \end{array}\right. \\
\mathsf{skip}
\end{array}
$$

and define an $\mathcal{L}$-program as a finite sequence of instructions such that the ‘loop’ and ‘end’ instructions occur in matching pairs.

Comparing $\mathcal{L}$ with $\mathcal{G}$, we find that

- ‘$V \leftarrow W$’ and ‘$V \leftarrow 0$’ are primitive instructions in $\mathcal{L}$, but not in $\mathcal{G}$ (*not an important difference*);
- ‘$V{-}{-}$’ is primitive in $\mathcal{G}$ but not in $\mathcal{L}$ (*also not important*);
- $\mathcal{L}$ has ***loops*** instead of ***labels*** and ***branches*** (*this is the important difference*!).

To complete our description of the $\mathcal{L}$-language, we give the precise meaning of the ***loop segment***

$$
\left\{\begin{array}{l} \mathsf{loop}\ V \\ \quad \mathcal{P} \qquad \} \text{ block} \\ \mathsf{end} \end{array}\right.
$$

Suppose that, when we read the ‘loop’ instruction, the value of $V$ is $v$. Then the block $\mathcal{P}$ of instructions is executed $v$ times—even if the value of $V$ is changed in $\mathcal{P}$. This means that ***$\mathcal{L}$-programs always halt***!

NOTE: The convention with respect to *input, output* and *auxiliary* variables is the same as before; i.e. all variables other than the *input* variables are initialised to 0.

EXAMPLES: $\mathcal{L}$-programs for ***addition*** and ***multiplication***:

$$\boxed{\begin{array}{l} Y \leftarrow X_1 \\ \textsf{loop } X_2 \\ \quad Y{+}{+} \\ \textsf{end} \end{array}} \quad \text{and} \quad \boxed{\begin{array}{l} \textsf{loop } X_1 \\ \quad \textsf{loop } X_2 \\ \quad\quad Y{+}{+} \\ \quad \textsf{end} \\ \textsf{end} \end{array}}\,.$$

## 10.2 Relationship to other notions of computability

Let $\mathcal{L}$-COMP be the class of functions computable by $\mathcal{L}$-programs.

**Lemma 10.1**. $\mathcal{L}\text{-COMP} \subseteq \mathcal{G}\text{-TCOMP}$.

**Proof:** Firstly, all $\mathcal{L}$-computable functions are ***total***.
Secondly, all $\mathcal{L}$-computable functions are $\mathcal{G}$-computable by the ***translation*** (or "***compilation***")

$$\mathcal{Q} \mapsto \mathcal{Q}'$$

of $\mathcal{L}$-programs into $\mathcal{G}$-programs, defined by CV induction on the lengths of programs $\mathcal{Q}$: $\boxed{V{+}{+}}$ and $\boxed{\textsf{skip}}$ are translated to themselves, and we have $\mathcal{G}$-macros for $\boxed{V \leftarrow 0}$ and $\boxed{V \leftarrow W}$. Finally, we can translate loop segments as follows:

$$\boxed{\begin{array}{l} \textsf{loop } V \\ \quad \mathcal{Q} \\ \textsf{end} \end{array}} \quad \longmapsto \quad \boxed{\begin{array}{ll} & Z \leftarrow V \\ [A] & \textsf{if } Z = 0 \textsf{ goto } E \\ & \mathcal{Q}' \\ & Z{-}{-} \\ & \textsf{goto } A \end{array}}$$

where $Z$ is a *new* (auxiliary) variable. $\square$

NOTE: We can easily define a *GN*, and hence an ***effective listing***, of $\mathcal{L}$-programs:

$$\mathcal{Q}_0,\ \mathcal{Q}_1,\ \mathcal{Q}_2, \ldots$$

Let $F_e$ be the unary function computed by $\mathcal{Q}_e$. Then

$$F_0,\ F_1,\ F_2, \ldots$$

is an ***effective enumeration*** of $\mathcal{L}$-COMP$^{(1)}$. Let

$$F(e,x) \ = \ F_e(x). \tag{1}$$

Then $F$ is ***total*** and clearly ***effective***, and hence (by CT) $\mathcal{G}$-computable. Hence by the method of Theorem 8.12,

$$\mathcal{L}\text{-COMP} \ \subset \ \mathcal{G}\text{-TCOMP} \tag{2}$$

with witness $\lambda x \cdot (F(x,x)+1)$.

The rest of this section is devoted to showing that

$$\mathcal{L}\text{-COMP} \ = \ \text{PR}.$$

**Lemma 10.2**. PR $\subseteq$ $\mathcal{L}$-COMP.

**Proof:** Suppose $f \in$ PR. We find an ***$\mathcal{L}$-program*** or ***macro*** for $f$ by ***CV induction on the length of a PR-derivation*** for $f$.
Consider the cases:

- The initial functions, i.e. the ***zero***, ***projection*** and ***successor*** functions, are computed by

$$\boxed{Y \leftarrow 0} \qquad \boxed{Y \leftarrow X_i} \qquad \text{and} \qquad \boxed{\begin{array}{l} Y \leftarrow X \\ Y{+}{+} \end{array}}\,.$$

- The $\mathcal{G}$-program for ***composition*** in the proof of Theorem 3.4 (p. 3-7) is also an $\mathcal{L}$-program.

- To get an $\mathcal{L}$-program for ***primitive recursion with parameters*** we must modify the method for Theorem 3.9 (pp. 3-8/9). Assuming $\mathcal{L}$-macros for $g$ and $h$, $f$ is computed by

```
Y ← g(X_1, ..., X_k)
loop X_{k+1}
   Y ← h(X_1, ..., X_k, Z, Y)
   Z++
end
```

The case of ***primitive recursion without parameters*** is similar. $\square$

In order to prove the converse of Lemma 10.2, we need certain definitions and lemmas.

Let $\mathcal{L}_n$ be the class of $\mathcal{L}$-programs with loop-end pairs nested to the depth of *at most* $n$, and $\mathcal{L}_n$-COMP the class of functions computed by $\mathcal{L}_n$-programs.

EXAMPLE: The program for *addition* is in $\mathcal{L}_1$, and for *multiplication* is in $\mathcal{L}_2$ (see previous example).

These definitions suggest a ***hierarchy of $\mathcal{L}$-programs***

$$\mathcal{L}_0 \subset \mathcal{L}_1 \subset \mathcal{L}_2 \subset \cdots , \qquad \mathcal{L} = \bigcup_{n=0}^{\infty} \mathcal{L}_n$$

and a ***hierarchy of $\mathcal{L}$-computable functions***

$$\mathcal{L}_0\text{-COMP} \ \subseteq \ \mathcal{L}_1\text{-COMP} \ \subseteq \ \mathcal{L}_2\text{-COMP} \ \subseteq \ \cdots ,$$
$$\mathcal{L}\text{-COMP} \ = \ \bigcup_n \mathcal{L}_n\text{-COMP}.$$

We assume for now:

- programs (or blocks) contain *only auxiliary variables* $Z_1$, $Z_2, \ldots$, and
- a block within a loop ('$\mathsf{loop}\ V\ \cdots\ \mathsf{end}$') does *not* contain the *loop variable* $V$. There is no loss of generality, since

$$\boxed{\begin{array}{l}\mathsf{loop}\ V\\ \quad \mathcal{Q}\\ \mathsf{end}\end{array}} \cong \boxed{\begin{array}{l} W \leftarrow V\\ \mathsf{loop}\ W\\ \quad \mathcal{Q}\\ \mathsf{end}\end{array}}$$

  where $W$ is a new auxiliary variable (and '$\cong$' denotes semantic equivalence of programs).

Now consider a block $\mathcal{P}$ with

$$\boldsymbol{var}(\mathcal{P}) \ \subseteq\ \vec{Z} \ \equiv\ \ Z_1, \ldots, Z_k.$$

We think of $\mathcal{P}$ as *transforming* the values of $\vec{Z}$ by

$$\begin{array}{rrcl} & \vec{z} & \mapsto & (f_1(\vec{z}), \cdots, f_k(\vec{z})) \\ \text{or} & \vec{z} & \mapsto & \vec{f}(\vec{z}) \end{array} \tag{3}$$

for certain $k$-ary functions $\vec{f} = f_1, \ldots, f_k$. We also say that

*$\mathcal{P}$ defines the transformation (3) on $\vec{Z}$.*

Now consider a loop segment

$$\mathcal{Q} \equiv \boxed{\begin{array}{l}\mathsf{loop}\ V\\ \quad \mathcal{P}\\ \mathsf{end}\end{array}}$$

with $V \not\equiv Z_i$ $(1 \leq i \leq k)$. Then $\boldsymbol{var}(\mathcal{Q}) \subseteq \{\vec{Z}, V\}$, and $\mathcal{Q}$ transforms the values of these variables by

$$\begin{array}{rcl} \vec{z} & \mapsto & \vec{g}(\vec{z}, v) \\ v & \mapsto & v \end{array} \tag{4}$$

for certain $(k+1)$-ary functions $\vec{g} = g_1, \ldots, g_k$ (since, by assumption, the value of the loop variable $V$ does not change with the execution of $\mathcal{Q}$). What is the relationship between $\vec{f}$ in (3) and $\vec{g}$ in (4)? Note that

> *$g_i(\vec{z}, v)$ is the final value of $Z_i$ after $v$ iterations of block $\mathcal{P}$, assuming that $v$ is the initial value of $V$.*

**Lemma 10.3**. *(With the above notation:) $\vec{g} \in \mathrm{PR}(\vec{f})$.*

**Proof:** We have

$$\begin{cases} g_i(\vec{z}, 0) &= z_i \\ g_i(\vec{z}, t+1) &= f_i(g_1(\vec{z}, t), \ldots, g_k(\vec{z}, t)). \end{cases}$$

So $\vec{g}$ is defined from $\vec{f}$ by *simultaneous primitive recursion*. The result follows from Theorem 6.6 (generalised to $k$ functions). $\square$

**Lemma 10.4**. *Suppose $\mathcal{P}$ is an $\mathcal{L}$-program with $\boldsymbol{var}(\mathcal{P}) \subseteq \vec{Z} \equiv Z_1, \ldots, Z_k$, and $\mathcal{P}$ defines the transformation*

$$\vec{z} \;\mapsto\; \vec{f}(\vec{z})$$

*with $\vec{f} = f_1, \ldots, f_k$. Then $\vec{f} \in \mathrm{PR}$.*

**Proof:** Since $\mathcal{P}$ is an $\mathcal{L}$-program, $\mathcal{P} \in \mathcal{L}_n$, for some $n$. We show that if $\mathcal{P} \in \mathcal{L}_n$ then $\vec{f} \in \mathrm{PR}$, by induction on $n$:

- **Basis:** $n = 0$. $\mathcal{P}$ has no loop-end pair, and consists only of the instructions

  $$\begin{array}{l} Z_i \leftarrow 0 \\ Z_i \leftarrow Z_j \\ Z_i{+}{+}. \end{array}$$

  So we must have

  $$\begin{array}{rl} & f_i(\vec{z}) = z_j + m \\ \text{or} & f_i(\vec{z}) = m, \end{array}$$

  for $i = 1, \ldots, k$, some $j$ and some $m$. Therefore $\vec{f} \in \mathrm{PR}$.

- **Induction step:** Suppose the result holds for $n$. Let $\mathcal{P} \in \mathcal{L}_{n+1}$. Then $\mathcal{P}$ is of the form

$$\begin{array}{l} \mathcal{Q}_0 \\ \textsf{loop } V_1 \\ \quad \mathcal{P}_1 \\ \textsf{end} \\ \mathcal{Q}_1 \\ \textsf{loop } V_2 \\ \quad \mathcal{P}_2 \\ \textsf{end} \\ \mathcal{Q}_2 \\ \ \vdots \\ \mathcal{Q}_{r-1} \\ \textsf{loop } V_r \\ \quad \mathcal{P}_r \\ \textsf{end} \\ \mathcal{Q}_r \end{array}$$

  where $\mathcal{Q}_i \in \mathcal{L}_0$ and $\mathcal{P}_i \in \mathcal{L}_n$. By the *induction hypothesis*, the transformations defined by these are all in PR. By Lemma 10.3, the transformation defined by

$$\boxed{\begin{array}{l} \textsf{loop } V_i \\ \quad \mathcal{P}_i \\ \textsf{end} \end{array}}$$

is PR. The result follows from the closure of PR under composition. □

We are ready to prove the converse of Lemma 10.2.

**Lemma 10.5**. $\mathcal{L}$-COMP $\subseteq$ PR.

**Proof:** Suppose the $k$-ary function $h$ is computed by the $\mathcal{L}$-program $\mathcal{P}$, containing the variables $Z_1, \ldots, Z_\ell$, $X_1, \ldots, X_k$, $Y$. Put

$$\begin{aligned} & \mathcal{P} \equiv \mathcal{P}(Z_1, \ldots, Z_\ell, X_1, \ldots, X_k, Y), \\ \text{and} \quad & \mathcal{Q} \equiv \mathcal{P}(Z_1, \ldots, Z_\ell, Z_{\ell+1}, \ldots, Z_{\ell+k}, Z_{\ell+k+1}), \end{aligned}$$

and suppose $\mathcal{Q}$ defines a transformation

$$\vec{z} \leftarrow \vec{f}(\vec{z})$$

with $\vec{z} \equiv z_1, \ldots, z_{\ell+k+1}$ and $\vec{f} = f_1, \ldots, f_{\ell+k+1}$. By Lemma 10.4, $\vec{f} \in$ PR. Finally

$$h(x_1, \ldots, x_k) = f_{\ell+k+1}(\underbrace{0, \ldots, 0}_{\ell \text{ times}}, x_1, \ldots, x_k, 0)$$

Therefore $h \in$ PR. $\square$

**Theorem 10.6**. $\mathcal{L}$-COMP $=$ PR.

**Proof:** By Lemmas 10.2 and 10.5. $\square$

**Corollary 10.7**. PR $\subset$ $\mathcal{G}$-TCOMP.

**Proof:** By (2) and Theorem 10.6.
(Cf. Thm 8.12, p. 807.) $\square$

NOTES:

1. Again, there is a ***relativised*** notion of '`loop`' computability, and a relativised version of Theorem 10.6:

$$\mathcal{L}\text{-COMP}(\vec{g}) = \text{PR}(\vec{g}) \tag{5}$$

2. We can define a ***relativised hierarchy***

$$\mathcal{L}_0(\vec{g}) \subset \mathcal{L}_1(\vec{g}) \subset \mathcal{L}_2(\vec{g}) \subset \cdots .$$

Then (see Thm 5.1 (p. 5-2) and Ex. 1 (p. 5-6))

$$\mathcal{L}_0(\vec{g}) \;=\; \text{ED}(\vec{g}).$$

### 10.3 Ackermann's function

As we have seen, the function $F$ in (1) (p. 10-3) is $\mathcal{G}$-computable, but not PR. We conclude this section with a ***more interesting*** and "***natural***" witness that PR $\subset$ $\mathcal{G}$-TCOMP. To set the stage, consider the hierarchy of PR definitions of well-known functions:

$$\begin{aligned}
x+0 &= x, & x+\boldsymbol{S}y &= \boldsymbol{S}(x+y)\\
x*0 &= 0, & x*\boldsymbol{S}y &= x+(x*y)\\
x\uparrow 0 &= 1, & x\uparrow \boldsymbol{S}y &= x*(x\uparrow y)\\
x\upuparrows 0 &= 1, & x\upuparrows \boldsymbol{S}y &= x\uparrow(x\upuparrows y)\\
&\vdots
\end{aligned}$$

NOTE 1: The ***hyperexponential***

$$x \upuparrows y = x^{x^{\cdot^{\cdot^{\cdot^{x}}}}} \Big\} \ (y \text{ times})$$

increases very rapidly with $y$.[1]

We systematise the above sequence of constructions by putting

$$f_1 = +,\ f_2 = *,\ f_3 = \uparrow,\ f_4 = \upuparrows, \ldots$$

and defining

$$\begin{cases}
f_0(x,y) &= \boldsymbol{S}y\\
f_{n+1}(x,0) &= \begin{cases} x & \text{if } n=0\\ 0 & \text{if } n=1\\ 1 & \text{if } n>1\end{cases}\\
f_{n+1}(x,\boldsymbol{S}y) &= f_n(x, f_{n+1}(x,y)).
\end{cases}$$

NOTES:

2. For all n, $f_n \in$ PR (by induction on $n$).
3. It is also easy to see that $f_n \in \mathcal{L}_n$-COMP (again by induction on $n$).
4. However, we can show that $f_{n+1} \notin \mathcal{L}_n$-COMP, since it "increases too rapidly"! (See [DW83], Chapter 13, for a proof for a related hierarchy.)

---

[1] For example, $3 \upuparrows 4$ is much larger than $10^{80}$, Eddington's estimate of the number of electrons in the universe.

Now define

$$\boldsymbol{A}(z, x, y) \;=\; f_z(x, y).$$

This is (a version of) ***Ackermann's function***.

NOTES:

5. The function $\boldsymbol{A}$ is defined by ***double recursion*** (on 1st and 3rd args):

$$\begin{cases} \boldsymbol{A}(0,\, x,\, y) \;=\; \boldsymbol{S}\, y \\ \boldsymbol{A}(\boldsymbol{S}\, z,\, x,\, 0) \;=\; \begin{cases} x & \text{if } z = 0 \\ 0 & \text{if } z = 1 \\ 1 & \text{if } z > 1 \end{cases} \\ \boldsymbol{A}(\boldsymbol{S}\, z,\, x,\, \boldsymbol{S}\, y) \;=\; \boldsymbol{A}(z,\, x,\, \boldsymbol{A}(\boldsymbol{S}\, z, x, y)). \end{cases}$$

6. $\boldsymbol{A} \in \mathcal{G}$-TCOMP (for example, by CT).

7. However, $\boldsymbol{A} \notin$ PR! For suppose

$$\boldsymbol{A} \in \mathrm{PR} = \mathcal{L}\text{-COMP} = \bigcup_n \mathcal{L}_n\text{-COMP}$$

Then for some $n$, $\boldsymbol{A} \in \mathcal{L}_n$-COMP. So

$$f_{n+1} = \lambda x, y \cdot \boldsymbol{A}(n+1,\, x,\, y) \in \mathcal{L}_n\text{-COMP},$$

a contradiction to Note 4.

EXERCISES:

1. Give an $\mathcal{L}$-program for the predecessor function.

2. Define the class $\mathcal{L}$-COMP$(\vec{g})$, and outline a proof for (5).

3. (***Tail recursion***.) Suppose $f$ is defined from $g$ and $h$ by the equations

$$\begin{cases} f(x, 0) \;=\; g(x) \\ f(x, n+1) \;=\; f(h(x, n), n). \end{cases}$$

Show that $f \in \mathcal{L}$-COMP$(g, h)$ and (hence) $f \in$ PR$(g, h)$. Note that in the "recursive call" (the expression on the right hand side of the second equation), $f$ is on the "outside"—this is characteristic of tail recursion. Also the ***parameter changes*** (from $x$ to $h(x, n)$), so that these equations (as they stand) do *not* form an instance of definition by primitive recursion.

4. Show that for sets: PR $\subset$ $\mathcal{G}$-COMP.