1 The idea of a register machine

Register machines present a model of computation much closer to the usual programming languages such as C. They consist of an arbitrarily large number of registers (although an individual program can make use of only finitely many of them), each of which contains a natural number, which are manipulated by a series of instructions (a program) which may use various types of branching or other control structures. We consider here two types of register machines, the Loop Program (or For Program) and the While Program. A third type, the Goto Program is discussed in a homework exercise.

2 Loop Programs

Consider the programming language which is built from the following:

1. Variables: $X_0$, $X_1$, $X_2$, etc. Formally, these can be named with finitely many symbols by using $X$, $X_1$, $X_{11}$, etc.

2. Assignment statements:
   
   (a) $X := 0$
   (b) $X := X + 1$
   (c) $X := X - 1$

3. For statements:

   for $X$ do $S$

   where $X$ is a variable and $S$ is a statement. This statement directs the machine to repeat the statement $S$ a number of times equal to the value of $X$ at the beginning of the loop, so if $X$ is initially 0 the statement is skipped. Note that $S$ may change the value of $X$, but this does not affect how many times $S$ is repeated (unlike, say, a C program); only the initial value of $X$ determines that. Also, the statement does not include a loop index, although one can be introduced by adding an additional variable which is initially set to 0 and is incremented by $S$.

4. Compound statements:

   begin $S_1; S_2; \ldots; S_n$ end

   where $S_1, \ldots, S_n$ are arbitrary statements.

A Loop Program (For Program) is a compound statement in this language. A program proceeds sequentially through the statements, with loops being iterated as described.
We say that a Loop Program computes an $n$-ary function $f$ if, when the program is started with the variables $X_1 = x_1, \ldots, X_n = x_n$, and all other variables 0, the program ends with the variable $X_0$ equal to $f(x_1, \ldots, x_n)$. We say that a function is Loop-computable if there is a Loop Program which computes it.

**Theorem 1** A function is Loop-computable if and only if it is primitive recursive.

In particular, all such functions are total (our structure does not allow infinite loops). One direction of the theorem is proved by induction on a primitive recursive derivation of $f$, and the other by induction on the construction of the program, starting with assignment statements and closing under for statements and compositions.

**Definition 2** The Loop Complexity of a Loop program is the maximum depth of a nested loop that occurs in the program. Formally, a program has loop complexity 0 if it consists only of assignment statements, and it has loop complexity $n + 1$ if it is a compound statement of the form begin $S_1; \ldots; S_n$ end where each $S_i$ is either a statement of loop complexity at most $n$ or is a loop statement of the form for $X$ do $S$ where $S$ has loop complexity $n$, and it contains at least one loop statement of this form.

### 3 While Programs

Consider the language with the following:

1. Variables, as before
2. Assignment statements, as before
3. **While** statements:
   
   \[ \text{while } X \neq 0 \text{ do } S \]

   where $X$ is a variable and $S$ is a statement.
4. Compound statements, as before

A **While Program** is a compound statement in this language. The conventions about computability are the same as with Loop Programs, with the exception that now a program will not necessarily halt on a given set of inputs, in which case the function is undefined for those values. A function is While-computable if there is a While Program which computes it (and hence halts precisely when the function is defined).

**Theorem 3** A partial function $f$ is While-computable if and only if it is partial recursive.

We will show that any partial recursive function is While-computable. The other direction will follow from later results about other models of computation. We proceed by induction on a partial recursive derivation. We start with the initial functions, and then close under composition, primitive recursion, and minimization:

First we note that we can use assignment statements of the form $X := Y$ as abbreviations for the following (where $Z$ is an otherwise unused variable):
begin
\[ X := 0; \ Z := 0 \]
while \( Y \neq 0 \) do begin \( X := X + 1; \ Z := Z + 1; \ Y := Y - 1 \) end
while \( Z \neq 0 \) do begin \( Y := Y + 1; \ Z := Z - 1 \) end
end

1. \( E(x) = 0 \). We use the program:
\[
\text{begin } X_0 := 0 \text{ end}
\]

2. \( S(x) = x + 1 \). We use the program:
\[
\text{begin } X_0 := X_1; \ X_0 := X_0 + 1 \text{ end}
\]

3. \( P^n_i(x_1, \ldots, x_n) = x_i \). We use the program:
\[
\text{begin } X_0 := X_1 \text{ end}
\]

4. Closure under composition. Let \( S_g \) compute the \( k \)-ary function \( g \) and \( S_1, \ldots, S_k \) compute the \( n \)-ary functions \( h_1, \ldots, h_k \). We describe a program to compute the composition \( f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_k(x_1, \ldots, x_n)) \). First pick a set of variables which are not used by any of the programs, \( Z_1, \ldots, Z_n, W_1, \ldots, W_k \). Note that a single loop allows us to perform assignment statements of the form \( X := Y \) as was done in the previous case; we will use such statements for simplicity. Our program then is:
begin
\[ Z_1 := X_1; \ldots \ Z_n := X_n \]
\( S_1 \)
\( W_1 := X_0 \)
\( X_1 := Z_1; \ldots \ Z_n := Z_n \)
\( S_2 \)
\( W_2 := X_0 \)
\[
\text{...}
\]
\( X_1 := Z_1; \ldots \ Z_n := Z_n \)
\( S_k \)
\( W_k := X_0 \)
\( X_1 := W_1; \ldots \ X_k := W_k \)
\( S_g \)
end
5. Closure under primitive recursion. Let $S_g$ and $S_h$ compute the functions $g$ and $h$. We compute $f(n, x_1, \ldots, x_k)$ as follows. For convenience, assume the inputs for $f$ are $N, X_1, \ldots, X_k$, for $S_g$ are $X_1, \ldots, X_k$, and for $S_h$ are $Z, N, X_1, \ldots, X_k$. Let $I, K, Z_1, \ldots, Z_k$ be unused variables.

\[
\begin{align*}
Z_1 &:= X_1; \ldots Z_k := X_k \\
S_g &

I &:= N \\
N &:= 0 \\
while I \neq 0 do \\
\begin{align*}
Z &:= X_0 \\
K &:= N \\
X_1 &:= Z_1; \ldots X_k := Z_k \\
S_h &

I &:= I - 1 \\
N &:= K \\
N &:= N + 1 \\
end
end
end
\]

6. Closure under minimization. Let $S_g$ compute $g(y, x_1, \ldots, x_n)$ with inputs $Y, X_1, \ldots, X_n$. We compute $f(x_1, \ldots, x_n) = \mu y[g(y, x_1, \ldots, x_n) = 0]$. Let $I, Z_1, \ldots, Z_n$ be unused variables.

\[
\begin{align*}
Z_1 &:= X_1; \ldots Z_n := X_n \\
I &:= 0 \\
X_0 &:= X_0 + 1 \\
while X_0 \neq 0 do \\
\begin{align*}
X_1 &:= Z_1; \ldots X_n := Z_n \\
Y &:= I \\
S_g &

I &:= I + 1 \\
end
I &:= I - 1 \\
X_0 &:= I \\
end
\]
4 Other alphabets

We have defined register machines on \( \mathbb{N} \), but we can also define them over an arbitrary alphabet \( A = A_k \). We need a few modifications to the above definitions. First, our assignment statements now have the form:

1. \( X := \emptyset \)
2. \( X := X \sigma_j \) for \( 1 \leq j \leq k \)
3. \( X := \text{Del}_r(X) \)
4. \( X := \text{Del}_l(X) \)

We need more complicated branching as well. For instance, in the case of While Programs, we need constructions of the form while \( X \neq Y \) do \( S \), and loops will depend on the length of \( X \). We will not discuss all the details of this here.