

DUE: Tue 19 February 2004. See URL for reading due dates.

1. Loop Programs

Loop programs are constructed from a primitive *basis* of statements of the form:

$$\{ X \leftarrow X + 1, X \leftarrow 0, X \leftarrow Y \}.$$

Each such primitive statement is a loop program. Given loop programs P_1 and P_2 , one forms new loop programs by the use of sequencing $P_1 ; P_2$ and the loop construct

```

loop X
  P1
end

```

In the above definitions, X and Y are any legitimate identifier names, and P_1 and P_2 are any legitimate loop programs.

Loop programs have the following semantics. Variables take on only values in the *natural numbers* $\mathbb{N} = \{0, 1, 2, \dots\}$, and a variable can hold an arbitrarily large natural number. The meaning of sequencing $P_1 ; P_2$ is the usual one. The meaning of the **loop** construct is that the code body P is executed exactly x times, where x is the value of X at the time the **loop** statement is first encountered. Loop programs are reminiscent of certain programming languages, such as Algol 68 and Ada, in which loop iteration bounds are evaluated *once*, at the time of loop entry, and are not able to be altered from within the loop body.

A loop program is formally defined as a triple $(P, I, O = \{Z\})$ consisting of the code P , a set $I = \{X_1, \dots, X_n\}$ of *input variables* and an *output variable* Z . (Sometimes more than one output variable is allowed in the *output set* O ; we will generally assume there is only one). If P is started with (non-negative integer) values x_1, \dots, x_n in the input variables and halts with z in the output variable Z , we say that z is computed by P from inputs x_1, \dots, x_n . (By convention, each uninitialized variable starts with the value zero). Thus each program with one output variable defines a function $f_P(x_1, \dots, x_n) : \mathbb{N}^n \rightarrow \mathbb{N}$.

The *nesting depth* of a loop program is the maximum level of loop nesting in the program, where zero-depth programs are taken to be those not using the **loop** construct (i.e., straight-line programs). For example

```

X ← 0 ;
loop Y
  Y ← Y + 1
end ;
loop Z
  X ← X + 1
end

```

has nesting depth 1, since the maximum depth of all concatenated loops is just 1.

Show how to compute each of the following functions by loop programs. Try to write loop programs of the smallest possible nesting depth.

- (a) $Z \leftarrow X + Y$, $Z \leftarrow X \dot{-} Y$ and for $Y > 0$, $Z \leftarrow \lfloor X/Y \rfloor$. Here $\lfloor a \rfloor$ is the truncation of a to an integer. The “monus” function $X \dot{-} Y$ is ordinary subtraction if $X \geq Y$ and otherwise evaluates to zero.

HINT: Before building the program for “monus”, build one that computes $Z \leftarrow X \dot{-} 1$. Believe it or not, you can get the effect of decrementing in a second variable Y by counting *upward*!! The answer is obvious when you finally get it ...

- (b) A loop predicate is a function computable by a loop program that returns either 0 or 1 for any inputs. Show that the following are loop predicates:

$sgn(X) \triangleq \text{if } X = 0 \text{ then } 0 \text{ else } 1$
 $\neg X \triangleq \text{if } X = 0 \text{ then } 1 \text{ else } 0$
 $X \vee Y \triangleq \text{if } X = Y = 0 \text{ then } 0 \text{ else } 1$
 $X = Y \triangleq \text{if } X = Y \text{ then } 1 \text{ else } 0$
 $X \leq Y \triangleq \text{if } X \leq Y \text{ then } 1 \text{ else } 0$

- (c) Show how a loop program can simulate the *conditional expression* $V \leftarrow \text{if } W = 0 \text{ then } X \text{ else } Y$.
- (d) Show how to use loop programs to simulate the "breakable loop" construct

```

loop X while Y  $\neq$  1
P
end

```

with a loop program having a nesting depth no more than 2 greater than that of P . In the above loop construct, the value of Y is alterable by execution of the loop body, but as usual the value of the loop limit X is fixed at loop entry. The loop terminates after X iterations or when $Y = 1$ —whichever comes first.

- (e) Write a **loop** program to compute the function $f(x) = \lfloor \log_2 x \rfloor$ for integer $x \geq 1$. *HINT*: Use the loop construct from (d) to make life easier.
- (f) Write a **loop** program to compute the function $g(x) = \lfloor \sqrt{x} \rfloor$ for integer $x \geq 0$.

GENERAL HINT: As you show how to implement harder and harder functions, like addition, multiplication, etc., express new functions in terms of old operators (this is in fact a kind of reduction.) That is, do NOT write out the implementational details each time in terms of the primitive basis $\{ ; \text{ loop } + 1 0 \}$. Imagine that each new operator that you implement, such as $Z \leftarrow X + Y$, is a "macro" that can later be automatically expanded into "native code" (the primitive basis), and program with these macros. The only rule is: make sure you have DEFINED an operator before you dare USE it. For example, one can implement $Z \leftarrow X * Y$ using **loop** \dots **end** and a "statement" like $Z \leftarrow U + V$. Don't write out all the details for the addition program again.

2. Diagonalization

At one time it was proposed to identify functions computable by loop programs (historically called *primitive* computable functions) with the notion of "algorithmically computable function". As this exercise shows, this proposal was doomed to defeat on the very reasonable grounds that the class of primitive computable functions is "incomplete"—there are functions that are clearly algorithmically computable and yet are *not* in the primitive computable class.

- (a) Prove by induction on the structure of loop programs that any loop program started with non-negative integer values in its input variables *always halts*. Therefore loop programs with n input variables realize *total* functions from \mathbb{N}^n to \mathbb{N} .
- (b) Show by a diagonalization argument that there are algorithmically computable functions from \mathbb{N} to \mathbb{N} that *cannot* be realized by any loop program. In other words, construct an effectively computable and *total* function on \mathbb{N} for which no loop program can be constructed. *HINT*: Each loop program, being a finite string, can be assigned an integer in an effectively computable way; this integer is its "Gödel number". From its Gödel number, the loop program can be effectively recovered. Use this effective correspondence to construct a function by diagonalization.

3. Non-Universality of the "Loop" Language

Refer to the above problems for a definition and properties of the loop programs. Let us think of this as a programming language named *LOOP*.

- (a) The question naturally arises: is *LOOP* universal? That is, can we write an interpreter for all *LOOP* programs in the language *LOOP*? Assume all *LOOP* programs are written in the ASCII character set. Then each program P can be considered as (encoded as) a string of zeros and ones, which can be considered the binary representation of an integer. Denote this unique integer by $\langle P \rangle$. It is the integer code or Gödel number of P . An interpreter U must be able to accept both a *LOOP* program $\langle P \rangle$ (encoded as an integer) and the appropriate number of inputs x_1, \dots, x_n (n equals the size of the input set I for P), and produce

the result $f_P(x_1, \dots, x_n)$ in the output register Z of U . (If P fails to halt for these inputs, an interpreter must also fail to halt for this program and these inputs). Thus U must faithfully imitate P on \bar{x} for any possible P and any \bar{x} .

Show that no interpreter for $LOOP$ can be written in $LOOP$. *HINT*: Diagonalization once again, but now for a different purpose ...

- (b) Of course anyone in this course *can* write an interpreter for $LOOP$ in \mathcal{C} . Explain why this does not contradict (a), even though you believe in Church's Thesis. What does it say about the difference between \mathcal{C} and $LOOP$?
- (c) You are aware by now that $LOOP$ supports loops that can be broken early, **for** loops, etc. Give an example of a looping construct that $LOOP$ *cannot* support, but that *is* supported in languages like \mathcal{C} , Pascal, etc. *HINT*: Recall from Problem 4 that all $LOOP$ programs halt on all inputs. What is it about the semantics of $LOOP$ loops that always guarantees this?

4. Turing Machine With Left Reset

A **TM with left reset** is similar to a standard (deterministic) TM except that the transition function has the form:

$$\delta: (Q - \{q_a, q_b\}) \times \Gamma \rightarrow Q \times \Gamma \times \{\mathbf{R}, \mathbf{RESET}\}.$$

If $\delta(q, a) = (p, X, \mathbf{RESET})$, then when the machine is in state q and reading an a , the machine overstrikes the current cell with X ; then the machine's head jumps to the leftmost cell at the left end of the tape, and enters state p . Note that these machines do not have the ability to move their head one cell to the *left* in a single move.

Show that Turing Machines with left reset recognize all and only the **CE** languages.

5. Turing Machine That Can Move Right or Stay

A **TM with stay-put instead of left** is similar to a standard (deterministic) TM except that the transition function has the form:

$$\delta: (Q - \{q_a, q_b\}) \times \Gamma \rightarrow Q \times \Gamma \times \{\mathbf{R}, \mathbf{STAY}\}.$$

If $\delta(q, a) = (p, X, \mathbf{STAY})$, then when the machine is in state q and reading an a , the machine overstrikes the a with X and the machine's head stays where it is (scanning the character X) and enters state p .

Note that these machines do *not* have the ability to move their head one cell to the left (*HINT*: No matter what they do they *cannot* go back to the left of the cell under scan.) At each point during computation the machine can overstrike the cell under scan and then move its head right one cell; or it can stay scanning the same cell and overstrike that cell under scan.

- (a) Show that this Turing Machine variant is *not* equivalent to the standard version, i.e., there are **CE** sets that such a machine *cannot* recognize.
- (b) [EXTRA CREDIT]: Exactly what class of languages do these machines recognize? Prove it.

6. Entertainment

Do not confuse "unsolvability" with "lack of information". Consider the following.

Let $A = \{s\}$ be a language containing only one string s , where the string is defined by:

$$s = \begin{cases} 1 & \text{if methane-based life exists on Titan} \\ 0 & \text{if not} \end{cases}$$

- (a) Is A decidable?
- (b) Why or why not? (The answer does *not depend in any way* upon your personal convictions about alien life forms.)