Solution Set 3

DATE: 29 SEPTEMBER, 1992

1. Problem 3.1.1: Devise an enumeration of while-programs based on base 22 number enumeration. Remark that there are 22 symbols we need to represent, see Table 1 in the text, page 46. Make a one-to-one correspondence of base 22 digits,

$$\{\, 0, 1, \ldots, 9, A, B, \ldots, L \,\},$$

and the symbols in our alphabet Σ we need to represent,

$$\Sigma = \{\, \text{begin, end, } \ldots, X, 0, \ldots, 9 \,\}.$$

The only subtlety of this problem is that a number is not exactly the same as a string of digits. Except for the number 0, no number begins with the digit 0. What to do? Various approaches are possible. Here is one: the map,

$$R : \mathbf{N} \longrightarrow \Sigma^*$$

from naturals to strings over the alphabet Σ is given by the formula: drop the leading digit, if no more digits, the string represented is the empty string, else do a 1-for-1 replacement of digits by symbols. This map R has every symbol string in its range, including the empty string. In fact, each string is represented at least 21 times. For instance,

```
begin
end
```

is represented by all of $101, 201, 301, \ldots, L01$.

2. Problem 3.1.2: Show that the encoding of while-programs by primes is one-to-one and give an algorithm which decides which numbers are in the image of the encoding. As many of you realized, the solution is obvious but very difficult to put into words. Also, many people didn't take into account that the algorithm must recurse on each exponent of a prime greater than 19! The problem is not clear about whether, for instance, 2 is in the range of the arithmetization. Previously only

programs enclosed in begin end were allowed. However, the most interesting issue here is whether we should allow recursion in our decoding algorithm. Note that this question is side-stepped on page 63 by presenting a non-recursive syntax checker. Chapter 6 returns to settle the question of the power of self-referential programming.

Suppose that $[P] = [Q]$, we show that $P = Q$. To apply induction to this problem, we make the following bizarre definitions. For $i = [I]$, some program I, define $k(i) = j$ where j is:

$$\max j \in \mathbf{N} \text{ such that either } 19^j | i \text{ or } 23^j | i.$$

Define $\kappa(i) = j$ where j is the unique integer such that j iterations of function k gives 0,

$$k(k(\ldots k(j)\ldots))(i) = 0.$$

For i coming from program I, the function $\kappa(i)$ is well-defined. That is, at most one of 19 or 23 can divide i. So the definition of k leaves nothing arbitrary. Therefore, the range of [], call it N_R, can be cut up into levels,

$$N_i = \{\, n \in N_R \,|\, \kappa(n) = i \,\}$$

and note that,

$$N_R = \cup_{i=1}^{\infty} N_i.$$

So suppose $[P] = [Q] \in N_1$. Then only rules (1) or (2) could have produced that common value. By the unique factorization of integers, $P = Q$. Suppose that if $[P] = [Q] \in N_i$ for $i < K$ then $P = Q$. We show that $[P] = [Q] \in N_K$ implies $P = Q$. Either rule (3) or (4) applies to the common value. If rule (3) applies, then by unique factorization of integers, $P = Q$ if the body of the while loop is the same. However, this body is an element of N_{K-1}, so by induction we are done. If rule (4) applies, again by unique factorization, we have reduced to the problem to finding if each statement in the begin-end clause decodes uniquely. Since each of these is an element of N_i for $i < K$, by induction we are done.

3. Problem 3.1.3: Deduce from the unsolvability of the Halting Problem that the function,

$$f(i) = \begin{cases} 1, & \text{if } \phi_i(0) \text{ converges;} \\ 0, & \text{if } \phi_i(0) \text{ fails to converge.} \end{cases}$$

is unsolvable. The intuition is that if f computable, we can trick it into making computable the Halting Problem. Since the Halting Problem is not computable, therefore neither can be f. The trick is to mutate any program P into a program P' which places a number e in $X1$ then runs P. So $P'(0)$ is the same as $P(e)$. Which e should we use? The e which is the index of P. This may be a bit confusing, but what we are talking about is a function $g : \mathbf{N} \to \mathbf{N}$ such that $\phi_e(e) = \phi_{g(e)}(0)$. In fact, $\phi_e e = \phi_{g(e)}(i)$ for any i.

What means $\phi_e(e) = \phi_{g(e)}(0)$? It means that either both are defined and equal or both are not defined. We shall show that such a g is computable. If f is computable, then $f \circ g$ is computable. But $f \circ g$ is the Halting Problem. Ooops! So f better not be computable.

Why is $f \circ g$ the Halting Problem?

$$f \circ g(i) = f(g(i)) = \begin{cases} 1 & \text{if } \phi_{g(i)}(0) \text{ converges;} \\ 0 & \text{if } \phi_{g(i)}(0) \text{ fails to converge.} \end{cases}$$

However, $\phi_{g(i)}(0) = \phi_i(i)$. Substituting this into the above, we conclude $f \circ g$ is the Halting Problem.

The loose ends to tie up are, 1) g is computable and, 2) given f and g computable, so is $f \circ g$. Here is a program which supposing `HALTONZERO` exists, would be the program `HALT`,

```
{HALT, 0 if program with index in X1 halts on }
{      input X1, else 1 }
{  assumes the function || which although it }
{  works on numbers, looks like con- }
{  catentation of symbols. The number (a constant!) }
{  for symbol string ‘‘abc’’ is writen ’abc’.}
begin
  {build P’ last symbol first...}
  X2 := ’end’ ; {X2 is now equal to 33}
  {now P’ will run P...}
  X2 := X1 || X2 ;
  {before running P, make X1 equal to X1}
  while X1<>0 do begin
```

```
    X2 := 'X1:=succ(X1);' || X2 ;
    X1 := pred(X1)
  end ;
  X2 := 'X1:=0;' || X2 ;
  X2 := 'begin' || X2 ;
  {place program P' as input to HALTONZERO}
  X1 := X2 ;
{does P' halt on zero?}
HALTONZERO
end
```