Computabilidad

Teoría de la programación I

Profs: Dina Wonsever
Juan José Prada

IN.CO.

Introducción.-

Se intentará dar respuesta a preguntas del tipo :

¿Todo problema tiene solución algorítmica ?

¿Toda función es computable ?

Se deben precisar las nociones de ***algoritmo*** y de ***ser computable***.

Algoritmo :

1) Descripción finita de un proceso computacional. Es un texto finito, escrito en un lenguaje en el cual cada sentencia tiene un significado no ambiguo.

2) Una operación básica (no coincide necesariamente con una sentencia del lenguaje) debe poder ser ejecutada en tiempo y espacio finito por un agente computacional.

3) Todo elemento de datos debe poder ser representado por una tira finita sobre un alfabeto finito.

O sea, podemos caracterizar un algoritmo como una descripción finita de las computaciones que se deben realizar sobre entradas para producir determinadas salidas. Esto equipara la idea de algoritmo a la de programa. (A veces se reserva el término *algoritmo* para procedimientos mecánicos que siempre paran y se utiliza la denominación *procedimiento efectivo* para aquellos que pueden no parar. Serían estos últimos entonces los asimilables a programas.

Un problema tiene solución algorítmica si existe un algoritmo que lo resuelve. En el caso de las funciones, diremos que una función es *computable* si existe un algoritmo que la computa.

Para llegar a resultados demostrados respecto a las preguntas iniciales se debe dar una caracterización formal de los conceptos intuitivos expuestos hasta el momento.

Para cualquier formalización que se elija, debe ser evidente que se ajusta a la noción intuitiva de *algoritmo* y de *computable*. Notar que la equivalencia entre la caracterización intuitiva y la formalización no se puede demostrar.

Existen varias formalizaciones alternativas :

- **Funciones recursivas** - 1931, Gödel, Herbrand
- **Máquinas de Türing** - 1936, Türing
- **λ-Cálculo** - 1936, Church
- **Lenguaje de prog.** - 1968, Engeler

Se ha demostrado la equivalencia entre todas estas formalizaciones (y otras que no se mencionaron). La "equivalencia" entre cualquiera de éstas y la noción intuitiva de algoritmo se conoce con el nombre de ***Tesis de Türing-Church***.

Se utilizará un lenguaje de programación elemental, el lenguaje P que se definirá más adelante, para el desarrollo de la teoría de computabilidad. A un programa en cualquier lenguaje le corresponde una función que asocia a cada valor de la entrada el correspondiente valor de la salida. Al concepto de problema resoluble algorítmicamente se le asocia entonces el de función computable.

Algunas definiciones relativas a funciones

Def.- Sean A y B dos conjuntos. f es una **función parcial** de A en B si :

- f es un subconjunto de A x B
- si $\langle x,y \rangle \in f$ y $\langle x,z \rangle \in f$ entonces y = z

Notación : f : A B

A - conjunto de definición
B - conjunto de valores

dominio - $dom(f) = \{ x \in A / \exists y \ \langle x,y \rangle \in f \}$
rango - $ran(f) = \{ x \in B / \exists y \ \langle y,x \rangle \in f \}$

Función total .- ($\rightarrow$) . El dominio es el conjunto de definición.

Función inyectiva.- f es inyectiva si $x \neq y$ implica $f(x) \neq f(y)$

Función sobreyectiva.- ran(f) = B

Teorema 1.- Sean A y B dos conjuntos no vacíos

1. Si $f : A \rightarrow B$ es una función total inyectiva, existe una función inversa de f total, sobreyectiva.

2. Si $g : B \rightarrow A$ es sobreyectiva, existe una función total inyectiva $f : A \rightarrow B$, tal que $g(f(x)) = x$

Dem.-
1) $A \neq \emptyset$, sea $a \in A$. Considerar la siguiente función :

$g(x) = y$ si $\exists\, y / f(y) = x$
a en caso contrario

- g es función por ser f inyectiva
- g es total (está definida para todo valor del argumento)
- se cumple $g(f(x)) = x$

2) Ejercicio

Conjuntos numerables

Def.- Un conjunto A es **numerable** si es vacío o si existe una función total sobreyectiva $f : N \rightarrow A$.

O sea, todo elemento de A ocurrirá por lo menos una vez en la secuencia
f(0), f(1), f(2), ...

Ejemplo.- Definamos Listas(N)

1. $[] \in$ Listas (N)

2. Si $n \in N$ y $L \in$ Listas(N) $\quad n.L \in$ Listas(N)

3. Todo elemento de Listas(N) se construye usando 1 y 2 un número finito de veces

Teorema 2.- Listas(N) es numerable.

Dem.- Por definición de conjunto numerable y por el teorema 1, basta con construir una función de Listas(N) en los naturales, total e inyectiva.

Sea la siguiente función $f : \text{Listas}(N) \rightarrow N$

1) si L es la lista vacía f(L) = 1
2) si L es la lista a1.a2.....an.[] , $f(L) = 2^{a1+1} 3^{a2+1} \ldots p_n^{an+1}$
donde p_n es el enésimo número primo

Como la descomposición en factores primos es única, dos listas distintas darán resultados distintos : f es inyectiva

Teorema 3 .- El conjunto de funciones totales $N \to N$ no es numerable.

Dem.- (Método de diagonalización de Cantor)

Supongamos que $N \to N$ es numerable. El conjunto obviamente no es vacío, de modo que debe existir una función total, sobreyectiva

$$f : N \to (N \to N)$$

Definamos una función $h : N \to N$, $h(x) = (f(x))(x) + 1$

h es total, debe existir un número n tal que

$f(n) = h$

Pero h difiere de todas las funciones f(i) en al menos un valor :

$h \neq f(0)$ ya que $h(0) = (f(0))(0) + 1 \neq (f(0))(0)$

.

.

$h \neq f(n)$ ya que $h(n) = (f(n))(n) + 1 \neq ((f(n))(n)$

Llegamos a una contradicción y debemos descartar la hipótesis realizada. $N \to N$ no es numerable

Teorema 4.- Si A es numerable y $B \subseteq A$, entonces B es numerable.

Dem.-

1) Si $B = \varnothing$, B numerable por def.

2) Sea $b \in B$
Como A es numerable , existe $f_A : N \to A$, sobreyectiva total

Definiremos $f_B : N \to B$

$$f_B(x) = \begin{cases} f_A(x) \text{ si } f_A(x) \in B \\ b \quad \text{en caso contrario} \end{cases}$$

f_B es total ya que está definida para todo natural

f_B es sobreyectiva ya que f_A lo es y $B \subseteq A$

Teorema 5 .- N ➥ N no es numerable

Dem. Inmediata por teorema anterior y por ser N → N no numerable.

Teorema 6 .- Si A es numerable, Listas(A) es numerable

Dem.- Existe una función total inyectiva g : A → N

Definiremos h : Listas(A) → N por

$h([]) = 1$

$h(a1.a2.....an.[]) = 2^{g(a1)}3^{g(a2)}...pn^{g(an)}$

h es total e inyectiva.

Un programa en un lenguaje de programación cualquiera es una secuencia finita de símbolos (lista de símbolos) sobre un alfabeto finito, o sea, sobre un conjunto numerable. Resulta entonces : el conjunto de los programas es numerable.

Hemos visto (Teorema 5) que el conjunto de las funciones parciales no es numerable. Por lo tanto, hay funciones parciales no computables.

LENGUAJE *P*

Hemos visto que el conjunto de funciones parciales computables es "más pequeño" que el conjunto de funciones parciales. Hemos establecido una relación de correspondencia entre funciones computables y programas. Precisaremos ahora que es un programa y que significa que un programa computa una función.

Para expresar el conjunto de programas definiremos un lenguaje de programación, el lenguaje ***P***. ***P*** es un lenguaje imperativo con cantidad mínima de construcciones. Lo definiremos dando :

1) La sintaxis de ***P***

2) La semántica de ***P***

3) La pragmática de ***P***

SINTAXIS DE *P*

Para presentar la sintaxis de ***P*** utilizaremos BNF (Backus-Naur form)

<prog> ::= PROGRAM (<var>) <sent> RESULT (<var>)

<var> ::= X<número>

<número> ::= <dígito> | <dígito><número>

<dígito> ::= 0 | 1 | ... | 9

<sent> ::= <assign> | <while> | <sec>

<assign> ::= <var> := <expr>

<expr> ::= 0 | SUC(<var>) | PRED(<var>)

<while> ::= WHILE <cond> DO <sent> END

<cond> ::= <var> ≠ 0

<sec> ::= <sent> ; <sent>

Vemos que el conjunto de los programas satisface :

$$\langle prog \rangle \subseteq \text{Lista(tokens)}$$

Definiremos la siguiente *"sintaxis abstracta" :*

- **Var** $\mathbf{E_1}$: $\dfrac{n \in \mathbf{n}}{var(n) \in Var}$

- **Cond** $\mathbf{E_1}$: $\dfrac{x \in Var}{x \neq 0 \in Cond}$

- **Expr** $\mathbf{E_1}$: $0 \in Expr$

  $\mathbf{E_2}$: $\dfrac{x \in Var}{suc(x) \in Expr}$

  $\mathbf{E_3}$: $\dfrac{x \in Var}{pred(x) \in Expr}$

- **Sent** $\mathbf{E_1}$: $\dfrac{x \in Var \quad e \in Expr}{assign(x,e) \in Expr}$

  $\mathbf{E_2}$: $\dfrac{t \in Cond \quad c \in Sent}{while(t,c) \in Sent}$

  $\mathbf{E_3}$: $\dfrac{c_1 \in Sent \quad c_2 \in Sent}{sec(c_1,c_2) \in Sent}$

- **Prog** $\mathbf{E_1}$: $\dfrac{x,y \in Var \quad c \in Sent}{prog(x,c,y) \in Prog}$

Ahora es fácil definir funciones sobre este nuevo conjunto, a los efectos de resolver problemas de caracter sintáctico.

Ejemplo :

Especificar una función que cuente cuantas veces aparece la variable **n** en un programa.

cant-veces : **N** x Prog $\rightarrow$ **N**

cant-veces (n,prog(x,c,y)) = $f_{Var}(n,x) + f_{Sent}(n,c) + f_{Var}(n,y)$

$f_{Var}(n,Var(m)) =$ 1 si n = m
0 en otro caso

$f_{Sent}(n,while(t,c)) = f_{Cond}(n,t) + f_{Sent}(n,c)$

$f_{Sent}(n,assign(x,e)) = f_{Var}(n,x) + f_{Expr}(n,e)$

$f_{Sent}(n,sec(c,d)) = f_{Sent}(n,c) + f_{Sent}(n,d)$

$f_{Cond}(n,Cond(x)) = f_{Var}(n,x)$

$f_{Expr}(n,0) = 0$

$f_{Expr}(n,suc(x)) = f_{Var}(n,x)$

$f_{Expr}(n,pred(x)) = f_{Var}(n,x)$

Esta función es, entonces, claramente computable.

Similarmente, se pueden especificar funciones como :

max-var-num : Prog $\rightarrow$ N	computa el mayor número de variable utilizada en un programa
cant-sent : Prog $\rightarrow$ N	computa la cantidad de sentencias de un programa

SEMÁNTICA DE *P*

Presentaremos una semántica operacional para ***P***. Se dará un conjunto de reglas precisas para determinar como es evaluada cada categoría sintáctica.

Las reglas van a ser de la forma :

1) $F \Rightarrow F'$ que se lee **F evalúa en F´**

y

2) $$\frac{G_1 \Rightarrow G_1' \quad ... \quad G_n \Rightarrow G_n'}{F \Rightarrow F'}$$

que se lee **F evalúa en F´**

si G_1 evalúa en G_1'

.

.

.

y G_n evalúa en G_n'

Como ***P*** es un lenguaje imperativo, será necesario definir la memoria de trabajo del programa. Dicha memoria contiene el valor de cada variable en un momento dado, o sea, el "estado".

Caracterizamos el estado como una *función parcial* entre las variables y los valores. Se tiene entonces:

$$\sigma \in \text{Estados} : \text{Var} \rightarrow \boldsymbol{N}$$

donde

$$\sigma = \{ <\text{var}_1 , \text{valor}_1 >,...\}$$

Sobre el conjunto de estados, se van a necesitar dos operaciones :

- valor_de : Var x Estados $\rightarrow \boldsymbol{N}$
- actualizar : Var x $\boldsymbol{N}$ x Estados $\rightarrow$ Estados

Para simplificar, se introducirá el estado Ω , para el cual toda variable tiene el valor 0.

Definiciones:

- valor_de(x,σ) = σ(x) el valor que toma x en el estado σ
- actualizar(x,n,σ) = σ′ donde
  σ′(y) = n si x = y
  σ(y) en caso contrario
- Ω(x) = 0 ∀ x ∈ Var

Abreviaciones:

- σ[x → n] ≡ actualizar(x,n,σ)
- σ[x → n, y → m] ≡ (σ[x → n])[y → m]

Notar que σ[x → n, x → m] = σ[x → m]

Para describir el valor de una expresión, necesitamos conocer en cual estado estamos. Describimos el valor por medio de una combinación de la expresión y el estado

$$< e, \sigma >$$

De la misma forma, se describe el valor de una sentencia, una condición y un programa por :

$$< c, \sigma > , < u, \sigma > \text{ y } < p,n >$$

donde, en el último caso, n es la entrada.

Estamos ahora en condiciones de definir las reglas para computar.

Comenzaremos por los programas.

$$\frac{<c,\Omega[x \to n]> \Rightarrow \sigma'}{<prog(x,c,y), n> \Rightarrow \sigma'(y)} \quad [P]$$

donde x, y ∈ Var , c ∈ Sent , n ∈ ***N*** y Ω, σ′ ∈ Estados

Para las sentencias, tenemos tres diferentes construcciones y por lo tanto, tres casos:

- Asignación : $\dfrac{<e,\sigma> \Rightarrow n}{<assign(x,e),\sigma> \Rightarrow \sigma[x \rightarrow n]}$ [A]

Donde los valores para las expresiones están definidos por las siguientes cuatro reglas :

- [E_1] $< 0,\sigma> \Rightarrow 0$
- [E_2] $< suc(x),\sigma> \Rightarrow suc(\sigma(x))$
- [E_3] $< pred(x),\sigma> \Rightarrow 0$ si $\sigma(x) = 0$
- [E_4] $< pred(x),\sigma> \Rightarrow n$ si $\sigma(x) = suc(n)$

- Secuencia : $\dfrac{<c,\sigma> \Rightarrow \sigma' \quad <d,\sigma'> \Rightarrow \sigma''}{<sec(c,d),\sigma> \Rightarrow \sigma''}$ [S]

- While : $\dfrac{<t,\sigma> \Rightarrow Falso}{<while(t,c),\sigma> \Rightarrow \sigma}$ [w_1]

$\dfrac{<t,\sigma> \Rightarrow Verd \quad <c,\sigma> \Rightarrow \sigma' \quad <while(t,c),\sigma'> \Rightarrow \sigma''}{<while(t,c),\sigma> \Rightarrow \sigma''}$ [w_2]

- Condición :
  - [V_1] $<cond(x),\sigma> \Rightarrow Falso$ si $\sigma(x) = 0$
  - [V_2] $<cond(x),\sigma> \Rightarrow Verd$ si $\sigma(x) =;/ 0$

Definición :

Decimos que hemos realizado una prueba para la semántica de un programa ***P*** cuando construimos un árbol de derivación, con la expresión para el valor del programa como consecuencia final (en "la base") y sin premisas por demostrar.

Ejemplo :

$$\frac{}{\langle SUC(X),\Omega[X\rightarrow 1]\rangle \Rightarrow 2}\,[E2]$$

$$\frac{\langle SUC(X),\Omega[X\rightarrow 1]\rangle \Rightarrow 2}{\langle ASSIGN(X,SUC(X)),\Omega[X\rightarrow 1]\rangle \Rightarrow \Omega[X\rightarrow 2]}\,[A]$$

$$\frac{\langle ASSIGN(X,SUC(X)),\Omega[X\rightarrow 1]\rangle \Rightarrow \Omega[X\rightarrow 2]}{\langle PROG(X,ASSIGN(X,SUC(X)),X),1\rangle \Rightarrow 2}\,[P]$$

Tenemos entonces un método para construir pruebas respecto al resultado (salida) de un programa dada la entrada.

Este método nos permitirá demostrar que un programa computa efectivamente una función dada.

Seremos más precisos en la noción de ***computable***.

Definición : Se dice que un ***P***-Programa **Q** ***computa*** la función parcial

$$\mathbf{P[Q]} : N \rightharpoonup N$$

donde

$$\mathbf{P[Q]}(m) = \begin{cases} n \text{ si } \exists\, n \in N \,/\, \langle \mathbf{Q},m\rangle \Rightarrow n \\ \text{indeterminado sino} \end{cases}$$

Definición : Una función parcial $f : \mathbf{N}\rightarrow N$ en ***P-computable***, sii existe un ***P***-programa que la computa.

Aplicaremos la tesis de Türing-Church, y diremos que si algo es computable, entonces es ***P***-computable.

Definición : Una computación de **Q** con una entrada **n** ***converge***, si

$$(\exists\, m \in N)\ \langle \mathbf{Q},\mathbf{n}\rangle \Rightarrow m$$

lo cual es denotado :

$\langle \overline{\mathbf{Q},\mathbf{n}}\rangle \downarrow$ o simplemente $\langle \mathbf{Q},\mathbf{n}\rangle \downarrow$

En caso contrario, se dice que ***diverge*** , y se anota de la siguiente manera :

$\langle \overline{\mathbf{Q},\mathbf{n}}\rangle \uparrow$ o simplemente $\langle \mathbf{Q},\mathbf{n}\rangle \uparrow$

Ejemplo :

Presentaremos un ejemplo más complejo, a los efectos de mostrar como se trabaja con esta técnica.

Que función computa el siguiente programa ***P***?

```
PROGRAM(X0)
        X1 := 0 ;
        WHILE X0 =;/ 0 DO
   c´       X1 := SUC(X1) ;
c           c" X0 := PRED(X0)
        END
RESULT(X1)
```

Como sólo X0 y X1 son usadas, se introduce la notación

$$\Phi(x,y) \equiv \Omega[X0 \rightarrow x, X1 \rightarrow y]$$

y para simplificar las cosas, supondremos que ya hemos demostrado :

Lema 1 :

$$(\forall\, x,y \in \boldsymbol{N}) \quad <c'',\Phi(s(x),y)> \Rightarrow \Phi\,(x,s(y))$$

Probaremos ahora :

Lema 2 :

$$(\forall\, x,y) \in \boldsymbol{N} \quad <c',\Phi(x,y)> \Rightarrow \Phi\,(0, x+y)$$

Probaremos este lema por inducción en x.

Paso Base: x = 0

Entonces por V1 y W1 se cumple que :

$$\frac{}{<X0=;/0,\Phi(0,y)> \Rightarrow \text{Falso}}\,[V1]$$

$$\frac{}{<c',\Phi(0,y)> \Rightarrow \Phi\,(0,y)}\,[W1]$$

$\Phi(0,y) = \Phi(0,x+y)$, por lo tanto el lema es válido para x=0.

Paso Inductivo: Sea x = s(n).

Hipótesis inductiva: $(\forall y \in \boldsymbol{N})$ $\langle c', \Phi(n,y)\rangle \Rightarrow \Phi(0,n+y)$

Por V2 se tiene (A) :

$$\frac{}{\langle X0=;/0, \Phi(s(n),y)\rangle \Rightarrow \text{Verd.}}\ [V2]$$

Por el Lema 1 (B) :

$$\frac{}{\langle c'', \Phi(s(n),y)\rangle \Rightarrow \Phi(n,s(y))}\ [Lema1]$$

Por lo tanto;

$$\frac{\dfrac{}{(A)\ (B)\quad \langle c', \Phi(n,s(y))\Rightarrow \Phi(0,n+s(y))}\ [HI]}{\langle c', \Phi(s(n),y)\rangle \Rightarrow \Phi(0,n+s(y))}\ [W2]$$

Pero $\Phi(0,n+s(y)) = \Phi(0,s(n)+y)$, entonces el lema se cumple para todo n.

El resto es simple :

$$\frac{\dfrac{\dfrac{}{\langle 0, \Omega[X0\rightarrow x]\rangle \Rightarrow 0}\ [E1]}{\langle X1=0, \Omega[X0\rightarrow x]\rangle \Rightarrow \Phi(x,0)}\ [Lema2] \qquad \langle c', \Phi(x,0)\rangle \Rightarrow \Phi(0,x)}{\langle X1:=0;\ c', \Omega[X0\rightarrow x]\rangle \Rightarrow \Phi(0,x)}\ [S]$$

$$\frac{}{\langle c,x\rangle \Rightarrow \Phi(0,x)(X1) = x}\ [P]$$

Por lo que se concluye que c es la función identidad.

PRAGMÁTICA DE *P*

Se utilizará el lenguaje P para escribir programas más complejos que los vistos hasta ahora. Por razones de comodidad, introduciremos nuevas construcciones, "abreviaciones" de secuencias de instrucciones en P. Se mantiene así la semántica que se le ha dado a ***P***.

Lo que haremos será introducir *macros* que se expandirán completamente en el texto del programa.

Se definirán macros para las sentencias, expresiones, condiciones y constantes.

Macros para las sentencias;

SENT ***abreviación*** **MACRO:**
-- declaraciones
<sent>
END

En las macros se utilizan macrovariables Y0 ,Y1 ,... las que en la expansión, serán sustituídas por una variable ***P***.

Ejemplo:

```
SENT  ASSIGN(Y1 ,Y2) MACRO:
 -- Y1:Var , Y2:Var
        Y1 := SUC(Y2) ;
        Y1 := PRED(Y1)
END
```

Al introducir macros, es un buen hábito probar que se comportan exactamente como queremos.

$$\frac{}{\langle SUC(y),\sigma\rangle \Rightarrow s(\sigma(y))}[E2] \qquad \frac{}{\langle PRED(x),\sigma[x\rightarrow s(\sigma(y))]\rangle \Rightarrow \sigma(y)}[E4]$$

$$\frac{}{\langle X:=SUC(y),\sigma\rangle \Rightarrow \sigma[x\rightarrow s(\sigma(y))]}[A] \qquad \frac{}{\langle X:=PRED(x),\sigma[x\rightarrow s(\sigma(y)]\rangle \Rightarrow \sigma'}[A]$$

$$\frac{}{\langle ASSIGN(x,y),\sigma\rangle \Rightarrow \sigma[x\rightarrow\sigma(y)]}[S]$$

donde $\sigma' = \sigma[x \rightarrow s(\sigma(y)), x \rightarrow\sigma(y)] = \sigma[x \rightarrow\sigma(y)]$

En el texto de la macros se puede usar también variables locales.

Ejemplo.

```
SENT IF  Y1=;/ 0  THEN Y2  FI  MACRO
 -- Y1: Var , Y2:Sent
 -- Y3: Var  (variable local)
        ASSIGN(Y3,Y1) ;
        WHILE Y3=;/ 0 DO
                Y2 ;
                Y3:= 0
        END
END
```

Las variables locales se cambian a variables con índices mayores que el mayor índice que haya en el programa (o mejor aún, en el programa expandido).

Las macro-definiciones para las expresiones son:

```
EXPR abreviación MACRO:
 -- declaraciones
        <sent>
RESULT (<macro-var>)
```

Ejemplo.

```
EXPR Y1+Y2 MACRO:
 -- Y1,Y2:Var
        WHILE Y2 =;/ 0 DO
                Y1:= SUC(Y1) ;
                Y2:= PRED(Y2)
        END
RESULT(Y1)
```

El tratamiento de las macros para las expresiones, es más complejo que el de las sentencias dado que se deben evitar cambios no deseados en los valores de las variables.

Por lo tanto, trataremos a las variables como variables locales inicializadas con las expresiones dadas en la macro-llamada.

Ejemplo:
Considere el siguiente programa.

```
PROGRAM (X1)
        X2:= X1+X1
RESULT(X2)
```

se expande en :

```
PROGRAM (X1)
  ASSIGN (X3,X1) ;  ---------- X3 := SUC(X1);
  ASSIGN (X4,X1) ;                X3 := PRED(X3);
  WHILE X4 =;/ 0  DO
        X3:= SUC(X3) ;
        X4:= PRED(X4) ;
  END;
  ASSIGN (X2,X3)
RESULT (X2)
```

Definiremos también macros para constantes, como por ejemplo;

```
EXPR  1  MACRO:
        Y1:= 0 ;
        Y1:= SUC(Y1)
RESULT (Y1)


EXPR  2  MACRO:
        Y1:= 1 ;
        Y1:= SUC(Y1)
RESULT (Y1)
```

etc.

Macros de Condición.

```
COND  "abreviación"  MACRO:
 -- declaraciones
        <sent>
RESULT (<macro-var>)
```

Ejemplo.

```
COND (Y1=;/ 0 ) & (Y2=;/ 0 ) MACRO
 --  Y1,Y2:Var;
        Y3:= 0 ;
        IF Y1=;/ 0 THEN
                IF Y2=;/ 0 THEN
                        Y3:= SUC(Y3)
                FI
        FI
RESULT (Y3)
```

Más adelante se van a usar:

- ES_PRIMO(n) → da como resultado 1 si **n** es primo y 0 si no es
- PRIMO(n) → para obtener el n-ésimo número primo
- PRIMEXP(m,n)→ devuelve el exponente del m-ésimo número primo en la descomposición en factores primos del número **n**

```
EXPR ES_PRIMO (Y0)  MACRO:
        Y1:= 1 ; Y2:= 1 ;
        WHILE Y2 =;/ 0 DO
                Y1:= SUC(Y1) ;
                Y2:= Y0 MOD Y1
        END ;
        Y3:= Y0 -Y1 ;
        IF Y3 =;/ 0 THEN
                Y4:= 0
        ELSE
                Y4:= 1
        FI
RESULT (Y4)
```

CODIFICACIÓN DE PROGRAMAS

Consideraremos computaciones con objetos distintos de los números naturales. Supongamos que tenemos una función

$g : A \rightarrow B$

La podremos computar si encontramos una función de codificación

$c : A \rightarrow N$

una función de decodificación

$d : N \rightarrow B$

y una función computable

$f : N \rightarrow N$

tal que

$g = d \circ f \circ c$

Los programas en ***P*** están restringidos a una sola entrada y una salida. Para manejar entradas o salidas múltiples codificaremos enuplas de naturales por naturales. Para lograr esto bastará con codificar pares.

$\langle x,y \rangle \in A \times B$ si $x \in A$, $y \in B$

fst (<a,b>) = a

snd (<a,b>) = b

Para codificar A x B definimos :

cod (<a,b>) = $2^{\{a\}}3^{\{b\}}$

(Usaremos a veces la notación <a,b> para referirnos a la codificación del par).

En lenguaje ***P*** tendremos las siguientes macros :

Para codificar :

```
EXPR PAR(Y1,Y2) MACRO
        Y3 := (2**Y1)*(3**Y2)
RESULT(Y3)
```

y para decodificar :

```
EXPR FST(Y1) MACRO
        Y2 := PRIMEXP(1,Y1)
RESULT(Y2)

EXPR SND(Y1) MACRO
        Y2 := PRIMEXP(2,Y1)
RESULT(Y2)
```

Construiremos programas en P cuyos datos de entrada y de salida son otros programas en P, eventualmente con sus datos.

Para eso necesitamos codificar y decodificar programas escritos en *P*. El conjunto de los programas *P* es un conjunto definido inductivamente; daremos funciones de codificación y decodificación para cada uno de los elemento sintácticos de *P*.

Las funciones de codificación tienen como dominio el conjunto del que se trate y como codominio los naturales. Las funciones de decodificación son funciones de naturales en el conjunto sintáctico del que se trate ; son , en general, funciones parciales.

Daremos a continuación los nombres de las funciones de codificación y decodificación y una expresión para las funciones de codificación.

VARIABLE :
cod-var (Xi) = i + 1
decod-var

CONDICION :
cod-cond (Xi /= 0) = i + 1
decod-cond

EXPRESION :
cod-expr(0) = 2
cod-expr(SUC(Xi)) = $3^{\{Xi\}}$
cod-expr(PRED(Xi)) = $5^{\{Xi\}}$
decod-expr

ASSIGN :
cod-ass (ass(x,e)) = $2^{\{x\}}\ 3^{\{e\}}$
decod-ass-var
decod-ass-expr

WHILE :
cod-wh(while(c,s)) = $5^{\{c\}}\ 7^{\{s\}}$
decod-wh-cond
decod-wh-sent

SECUENCIA :
cod-sec(sec(s1,s2)) = $11^{\{s1\}}\ 13^{\{s2\}}$
decod-sec-1
decod-sec-2

Dado que (s1;s2);s3 es, de acuerdo a la semántica, equivalente a s1;(s2;s3) se implementa cod-sec de modo que en ambos casos la codificación sea :

$$11^{\{s1\}}\ 13^{\{s2;s3\}}$$

donde s1 no es una secuencia

PROGRAMA :
cod-prog(prog(x,c,y)) = $2^{\{x\}}\ 3^{\{c\}}\ 5^{\{y\}}$
decod-prog-vare
decod-prog-sent
decod-prog-vars

Por ejemplo, al siguiente programa *R*

```
PROGRAM(X0)
        X0 := 0
RESULT(X0)
```

le corresponde el número

$$2.3^{2.9}.5 = 10.3^{18}$$

Las funciones de codificación y decodificación se implementan mediante macros en ***P*** (de tipo expresión).

En algún caso nos interesará chequear la pertenencia de un elemento de nuestro lenguaje a determinada categoría sintáctica. Esto se traduce en macros de condición :
es-0, es-pred, es-suc
es-ass, es-while, es-sec

Llamaremos índice de un programa al número que le corresponde en la codificación.

Dado el índice **q**

Ix(q) - programa de índice q Ix : N $\rightarrow$ Prog

ϕq - función computada por el programa Ix(q)

$\phi q(m) = n$ sii $<Ix(q),m> \Rightarrow n$

Dado que no todo natural es el índice de algún programa, para que Ix sea una función total definimos :

Ix(q) = (PROGRAM(X0) X0 := 0 RESULT(X0)) si q no es el índice de ningún programa.

UN INTÉRPRETE PARA *P* EN *P*

Construiremos en *P* un programa IP que , recibiendo como entrada un programa en *P* Q (su codificación) y un natural m, realiza la ejecución de Q con entrada m, devolviendo el resultado correspondiente.

IP utiliza las reglas de evaluación de las instrucciones de *P*, o sea, la semántica operacional que hemos definido para el lenguaje *P*.

Definiremos previamente un método para codificación del estado de un programa y las correspondientes funciones de acceso y actualización

El estado es una función de variables en naturales

$\sigma : \{Var\} \rightarrow N$

cod-estado$(\sigma) = 2^{\sigma(X0)}.3^{\sigma(X1)}.5^{\sigma(X2)}. \ldots$

esto implica cod-estado(Ω) = 1

valor-de (x,σ) se implementa mediante PRIMEXP y

actualizar(x,n,σ) mediante PRIMEXP y operaciones aritméticas.

Notar que si bien un estado es un conjunto infinito (cantidad infinita de variables), logramos codificar un estado mediante un natural, aprovechando el hecho de que sólo una cantidad finita de variables puede tener valor distinto de 0.

Las siguientes macros , además de las ya mencionadas,serán utilizadas :

eval-ass(sent-ass,estado) - modifica el estado
eval-cond(cond,estado) - evalúa en un booleano
es-sec(sent) - chequea si una sentencia es una secuencia
es-ass(sent) - chequea si una sentencia es una asignación
cod-sec(s1,s2) - codifica una secuencia a partir de las sentencias s1 y s2

La entrada a IP es la codificación de un par <q,n> ; se cumple:

$< IP,\{<q,n>\} > \Rightarrow m$ si $< Ix(q),n > \Rightarrow m$ y

$< IP,\{<q,n>\} > \uparrow$ si $< Ix(q),n > \uparrow$

IP :

```
PROGRAM(X0)
     X1 := FST(X0);       X1 - programa Q   prog(x,c,y)
     X2 := SND(X0);       X2 - entrada n a Q
     X3 := DECOD-PROG-VARE(X1);
     X4 := DECOD-PROG-SENT(X1);
     X5 := DECOD-PROG-VARS(X1);
     X6 := OMEGA;                          X6 - estado
     X6 := ACTUALIZAR(X3,X2,X6);
     WHILE X4 ≠ 0 DO
          IF ES-SEC(X4)
          THEN X7 := DECOD-SEC-1(X4);
               X4 := DECOD-SEC-2(X4)
          ELSE X7 := X4;
               X4 := 0
          FI;
          IF ES-ASS(X7)
          THEN X6 := EVAL-ASS(X7,X6)
          ELSE X8 := DECOD-WH-COND(X7);
               X8 := EVAL-COND(X8,X6);
               IF X8 = 1
               THEN X9 := DECOD-WH-SENT(X7);
                    IF X4 ≠ 0
                    THEN X4 := COD-SEC(X7,X4)
                    ELSE X4 := X7
                    FI;
                    X4 := COD-SEC(X9,X4)
               FI
          FI
     END;
     X10 := VALOR-DE(X5,X6)
RESULT(X10)
```

Asociada al programa IP podemos construir la macro MIP, para manejar IP como una expresión. Utilizaremos a menudo la macro eval-prog(p,n) que codifica el par <p,n> y evalúa Ix(p) con entrada n mediante la macro MIP.

```
EXPR EVAL-PROG (Y0,Y1) MACRO
     Y2 := COD-PAR(Y0,Y1);
     Y3 := MIP(Y2)
RESULT(Y3)
```

En algunas situaciones necesitaremos utilizar la expresión :

eval-prog-step(p,n,m) = <1,w> si <Ix(p),n> $\Rightarrow$ w en a lo sumo **m** pasos
= <0,0> en otro caso

Para poder construir una macro en *P* debemos modificar el intérprete de modo que evalúe el programa que se interpreta una cantidad dada de pasos.

Intérprete modificado - IPS

```
PROGRAM(X0)                X0 - <p,<n,m>>
     X1  :=  FST(X0);                X1  -  programa  Q
prog(x,c,y)
     X2 := SND(X0);        X2 - <n,m>
     X11 := SND(X2);       X11 - m
     X2 := FST(X2);        X2 - n
     X3 := DECOD-PROG-VARE(X1);
     X4 := DECOD-PROG-SENT(X1);
     X5 := DECOD-PROG-VARS(X1);
     X6 := OMEGA;                        X6 - estado
     X6 := ACTUALIZAR(X3,X2,X6);
     WHILE X4 ≠ 0  & X11 ≠ 0  DO
          IF ES-SEC(X4)
          THEN X7 := DECOD-SEC-1(X4);
               X4 := DECOD-SEC-2(X4)
          ELSE X7 := X4;
               X4 := 0
          FI;
          IF ES-ASS(X7)
          THEN X6 := EVAL-ASS(X7,X6)
          ELSE X8 := DECOD-WH-COND(X7);
               X8 := EVAL-COND(X8,X6);
               IF X8 = 1
               THEN X9 := DECOD-WH-SENT(X7);
                    IF X4 ≠ 0
                    THEN X4 := COD-SEC(X7,X4)
                    ELSE X4 := X7
                    FI;
                    X4 := COD-SEC(X9,X4)
               FI
          FI;
     X11 := PRED(X11)
     END;
     IF X4 ≠ 0
     THEN X10 := COD-PAR(0,0)
     ELSE X10 := VALOR-DE(X5,X6);
          X10 := COD-PAR(1,X10)
     FI
RESULT(X10)
```

Mediante la macro MIPS asociada a IPS podemos construir la macro eval-prog-step en modo análogo a eval-prog

ALGUNAS FUNCIONES NO COMPUTABLES

Diremos que una función f : N → N es **computable** cuando existe un programa en P, Q, de índice q, tal que $\phi_q = f$

Consideremos la función θ : N → {0,1} definida por :

θ(n) = 1 si <Ix(n),n> ↓
= 0 si <Ix(n),n> ↑

Teorema 7 .- θ no es computable

Demostraremos este resultado por contradicción.

Supongamos que θ es computable. Existe entonces un programa Q que computa θ . Sea QM la macro de expresión asociada. Consideremos el siguiente programa *R* :

```
PROGRAM(X0)
     X1 := QM(X0);
     WHILE X1 ≠ 0 DO SKIP END
RESULT(X0)
```

(SKIP es una macro de sentencia que no afecta el estado).

Consideremos la propiedad de terminación de *R*

1) *R* diverge con entrada n si QM(n) da un valor distinto de 0.
Pero QM(n) ≠ 0 sii n representa un programa que converge con entrada n.
Por lo tanto :

<R,n> ↑ si <Ix(n),n> ↓

2) *R* converge con resultado 0 para la entrada n si QM(n) da el valor 0. Pero QM(n) = 0 sii n representa un programa que diverge con entrada n. Por lo tanto :

<R,n> ↓ si <Ix(n),n> ↑

Sea r el índice del programa R, Ix(r) = R

Consideremos que pasa con R cuando la entrada es r

1) <R,r> ↑ si <Ix(r),r> ↓ , o sea si <R,r> ↓

2) <R,r> ↓ si <Ix(r),r> ↑ , o sea si <R,r> ↑

Como un programa P para una entrada debe converger o diverger, hemos llegado a una contradicción y debemos descartar la única hipótesis : que θ es computable.
Por lo tanto, θ no es computable !!

Necesitamos eval-prog-step para resolver algunos problemas "semidecidibles". Consideremos la función parcial :

$f(q,r,n) = 1$ si $\phi_q(n) = 1$ o $\phi_r(n) = 1$
indefinida en caso contrario

Si tratamos de usar eval-prog para computar esta función, tendremos el problema de que la computación puede no terminar para el primero de los programas que se evalúa, en un caso en que sí termina el segundo.

Resolveremos este problema mediante el siguiente programa:

```
PROGRAM(X0)
     X1 := FST(X0);       q
     X2 := SND(X0);       <r,n>
     X3 := SND(X2);       n
     X2 := FST(X2);       r
     X4 := 1;  X5 := 1;
     WHILE X4 ≠ 0 DO
          X6 := EVAL-PROG-STEP(X1,X3,X5);
          IF SND(X6) = 1 THEN X4 := 0 FI;
          X6 := EVAL-PROG-STEP(X2,X3,X5);
          IF SND(X6) = 1 THEN X4 := 0 FI;
          X5 := SUC(X5)
     END;
     X4 := 1
RESULT(X4)
```

Consideremos la siguiente función

stop (p,n) = 1 si $\langle Ix(n),n\rangle \downarrow$
0 en caso contrario

Teorema 8.- stop no es computable

Demostraremos este teorema utilizando el resultado ya conocido de que θ no es computable. La técnica de reducir nuestro problema a un problema que ya sabemos que no es computable se usará muchas veces.
1) Supondremos que la función es computable
2) Utilizando el supuesto programa para computarla escribimos un programa que computa una función que ya sabemos no es computable.

Supongamos que stop es computable. Existe una macro MSTOP que computa la expresión asociada a la función stop. Podemos construir entonces el siguiente programa :

```
PROGRAM(X0)
     X1 := MSTOP(X0,X0)
RESULT(X1)
```

Pero este programa computa la función θ !!

Consideraremos una clase más restringida de programas y veremos si se siguen manteniendo los resultados sobre no computabilidad obtenidos hasta el momento. Se trata de los ***programas constantes***. Los definiremos como aquellos que no usan su variable de entrada.

Es claro que estos programas computan una función constante; si bien hay otros programas que pueden computar una función constante, sólo serán considerados los que no utilizan la variable de entrada en ninguna sentencia. Esta es una cuestión sintáctica acerca de un programa y es computable , la condición que se debe cumplir es :
cant-veces(x,prog(x,c,y)) = 1).

Antes de probar que la función stop restringida a programas constantes no es computable introduciremos la función $g : NxN \rightarrow N$ t.q.

Dado q índice del programa Q ≡
```
PROGRAM(Xi)
    C
RESULT(Xj)
```

y n, valor de su entrada, g(q,n) = q´ tal que

Ix(q´) es
```
PROGRAM(Xmax+1)
     Xi := n;
     C
RESULT(Xj)
```

O sea, g es una función que transforma programas en programas (índices de programas en índices de programas,por supuesto). El programa resultado es un programa constante de acuerdo a nuestra definición.

Notar que :
$\phi_{g(q,n)}(m) = \phi_q(n)$ para todo m natural

g es computable, sea GM la macro que la computa

```
EXPR GM(Y1,Y2)  MACRO
     Y3 := DECOD-PROG-VARE (Y1);
     Y4 := DECOD-PROG-SENT (Y1);
     Y5 := DECOD-PROG-VARS (Y1);
     Y6 := MAX-VAR-NUM (Y1);   Y6 := SUC(Y6);
     Y7 := COD-EXPR-0;
     Y7 := COD-ASS(Y3,Y7);
```

```
        IF Y2 ≠ 0 THEN
             Y8 := COD-SUC(Y3);
             Y8 := COD-ASS(Y3,Y8);
             Y9 := COD-SEC(Y8,Y4);
             Y2 := PRED(Y2);
             WHILE Y2 ≠ 0 DO
                  Y9 := COD-SEC(Y8,Y9);
                  Y2 := PRED(Y2)
             END;
             Y9 := COD-SEC(Y7,Y9)
        ELSE
             Y9 := COD-SEC(Y7,Y4)
        FI;
        Y10 := COD-PROG(Y6,Y10,Y5)
    RESULT(Y10)
```

Teorema 9 .- stopK(p) = 1 si Ix(p) es un programa cte y <Ix(p),m> $\downarrow \forall$ m

0 en caso contrario

no es computable.

Dem.- Supongamos que stopK es computable; sea MSK la macro que la computa. Utilizando además la macro GM podemos construir la siguiente macro :

```
EXPR Q(Y0,Y1) MACRO
        Y2 := GM(Y0,Y1);
        Y2 := MSK(Y2)
RESULT(Y2)
```

Pero esta macro computa stop(p,n). Dado que la función g es computable, como lo demuestra la macro GM que hemos construido, debemos descartar la existencia de MSK, o sea, stopK no es computable.

El problema del ***código muerto***.-

Supongamos que en un programa tenemos :

```
...
...
X1 := 0;
IF X1 ≠ 0 THEN C FI;
...
```

Sabemos que el conjunto de sentencias C nunca será ejecutado.

Es posible para un computador averiguar si existe ***código muerto*** en un programa?

Para representar adecuadamente este problema numeraremos las sentencias del programa, p.ej.

```
1       X := ..
2       WHILE ...
3               X1 := ..
4               X2 := ..
        END;
5       X6 := ..
        .
        .
```

Definamos el predicado

ejec-sent-num (p,n,m) verdadero si la sentencia número m es ejecutada alguna vez en la computación de Ix(p) con entrada n

y la función

$$\delta\,(p,m) = \begin{cases} 1 & \text{si } \exists\, n \,/\, \text{ejec-sent } (p,n,m) \text{ verdadero} \\ 0 & \text{en caso contrario} \end{cases}$$

Entonces, si $\delta\,(p,m) = 0$, la sentencia m es código muerto en Ix(p).

Nos interesa saber si $\delta\,(p,m)$ es computable.

Veremos primero que pasa con un caso más simple :

$$\delta'(p,n,m) = \begin{cases} 1 & \text{si ejec-sent-num } (p,n,m) \text{ verdadero} \\ 0 & \text{en caso contrario} \end{cases}$$

Esta función no es computable.

Consideremos una macro PM , que dado un programa Q con índice **q** y **max** como número más alto de variable :

```
PROGRAM(Xi)
        C
RESULT(Xj)
```

calcula el par <p,m>, donde p es el índice del programa :

```
PROGRAM(Xi)
        C;
        Xmax+1 := 0
RESULT(Xj)
```

y m el número de la última sentencia de este programa.

(Ejercicio, construir la macro PM).

Supongamos que δ' es computable. Sea MDELTA una macro que la computa. Podemos construir el siguiente programa :

```
PROGRAM(X0)                 X0 = <q,n>
     X1 := FST(X0);         q
     X2 := SND(X0);         n
     X3 := PM(X1);          <p,m>
     X4 := FST(X3);         p
     X5 := SND(X3);         m
     X6 := MDELTA(X4,X2,X5)
RESULT(X6)
```

Este programa computa stop(q,n) !!

(Notar que los programas en P terminan siempre su ejecución en la última sentencia sintáctica del programa y que se cumple:

$\langle Ix(q),n\rangle \downarrow$ sii Ix(p) ejecuta la sentencia m al computar con entrada n)

Ejercicio.- Demostrar que $\delta(p,m)$ no es computable.

Otra cuestión de interés práctico es si existe la posibilidad de saber si dos programas computan la misma función.

Trataremos primero un caso más simple.

Consideremos la función :

$$\mathbf{eq_{id}(x)} = \begin{cases} 1 & \text{si } \phi_X = id \text{ (la identidad)} \\ 0 & \text{en caso contrario} \end{cases}$$

Teorema 10 .- eq_{id} no es computable.

Dem.- Dado un programa q y un número n podemos computar el índice del programa Q :

```
PROGRAM(X0)
     X1 := EVAL-PROG(q,n)
RESULT(X0)
```

Q computa la identidad sii $\langle Ix(q),n\rangle \downarrow$

Supongamos que eq_{id} es computable, sea MID una macro que la computa. Sea MQ una macro que dados q y n calcula el índice del programa Q anterior. Podemos construir :

```
PROGRAM(<XO,X1>)
     X2 := MQ(X0,X1);
     X3 := MID(X2)
RESULT(X3)
```

Este programa computa stop (q,n) !!

Teorema 11 .- No existe un programa que pueda decidir si dos programas arbitrarios computan la misma función.

Dem.- Ejercicio . Se sugiere prueba por contradicción utilizando el teorema 10.

CONJUNTOS RECURSIVAMENTE ENUMERABLES

Def.- Un conjunto $A \subseteq N$ es recursivamente enumerable (r.e.) si

$A = \varnothing$ o

existe una función P-computable, sobreyectiva, total

$ef : N \rightarrow A$

Esto significa que por medio de una máquina podemos encontrar todo elemento del conjunto contando :

ef(0), ef(1), ef(2), ...

Si deseamos demostrar que un conjunto A es r.e. escribimos un programa ***P*** EFA que :

1) Siempre termina
2) El resultado está siempre en A
3) Todo elemento de A es el resultado de <EFA,n> para algún n

Decimos que EFA **genera** el conjunto A

Algunos conjuntos r.e. :

- N
- $\{n \in N / n \text{ es primo } \}$
- conjuntos definidos inductivamente
- lenguaje generado por una gramática

A partir de la definición se deduce inmediatamente :

Teorema 12.- Un conjunto r.e. es numerable.

Teorema 13.- La unión y la intersección de dos conjuntos r.e. es también r.e..

Dem.- Lo demostraremos para la unión.

A r.e. , B r.e. $\rightarrow$ A U B es r.e.

- Si ambos conjuntos son vacíos, la unión es vacía y se cumple por def.

- Si uno de los dos conjuntos es vacío, la unión coincide con el otro, y es r.e. por hipótesis

- Supongamos que ninguno de los dos conjuntos es vacío :

Como A es r.e. , existe EFA : $N \rightarrow A$, sobreyectiva total
Como B es r.e. , existe EFB : $N \rightarrow B$, sobreyectiva total

Construiremos un programa para EF(AUB)

```
PROGRAM(X0)
     X1 := X0 MOD 2;
     X2 := XO DIV 2;
     IF X1 /= 0 THEN
          X3 := EFA(X2)
     ELSE X3 := EFB(X2)
     FI
RESULT(X3)
```

Observemos que para el programa anterior se cumple :

1) Siempre para

2) El resultado está en A o en B, o sea, está en A U B

3) Consideremos un elemento cualquiera x de A U B

- $x \in A$, existe n t.q. EFA(n) = x, se cumple :
  EF(AUB)(2n+1) = x

- $x \in B$, existe m t.q. EFB(m) = x, se cumple :
  EF(AUB)(2m) = x

Por lo tanto, A U B es r.e..

Ejercicio.- Demostrar el resultado para la intersección.

- Un conjunto numerable y no r.e.

La mayoría de los conjuntos numerables que hemos visto es también r.e.. Consideremos un conjunto que no lo es :

$TOT = \{ i \in N / \phi_i \text{ es total } \}$

TOT es numerable, ya que está contenido en el conjunto de programas.

Teorema 14.- TOT no es r.e.

Dem.- Supongamos que TOT es r.e. Claramente TOT no es vacío, entonces debiera existir una función de enumeración :

$$EFTOT : N \rightarrow TOT$$

Consideremos la siguiente función :

$$g(x) = EFTOT(x)(x) + 1$$

g es computable, ya que EFTOT lo es
g es total , por razón similar

pero g difiere de toda función en la enumeración (por construcción, difiere en al menos un valor)
Por lo tanto, TOT no es r.e.

Teorema 15.- Un conjunto A es r.e. sii existe una función computable, parcial
f : N ➥N , tal que A = dom(f)

Dem.
⇒) Supongamos que A es r.e.
Si A = **Φ** , f es la función vacía

En caso contrario, considerar :

```
PROGRAM(X0)
     X1 := 0;
     WHILE EFA(X1) ≠X0 DO X1 := SUC(X0) FI
RESULT(X1)
```

Claramente este programa entra en loop sii la entrada no pertenece a A. Sea f la función computada por este programa. Se cumple : dom(f) = A.

⇐) Sea f : N ➥ N , computable, dom(f) = A
Sea q el índice del programa que computa f

Si A = ∅ , A es r.e. por definición

Supongamos A ≠ ∅ , a ∈ A , consideremos el siguiente programa :

```
PROGRAM(<X1,X2>)
     X3 := EVAL-PROG-STEP (q,X1,X2);
     IF FST(X3) ≠ 0
          THEN X5 := X1
          ELSE X5 := a
     FI
RESULT(X5)
```

1) Este programa siempre para (recordar que eval-prog-step siempre para)

2) El resultado está en A. En efecto, el resultado es , o bien a que pertenece a A, o bien, un elemento de dom(f). (Recordar que Ix(q) computa f).

3) Para todo elemento de A existe alguna entrada que lo genera.
En efecto, si $x \in A$, $x \in dom(f)$, existe un natural m tal que Ix(q) para frente a x en a lo sumo m pasos. El programa anterior, con entrada <x,m>, da como resultado x.

Este resultado nos lleva a otro modo de caracterizar un conjunto r.e.

Teorema 16.- Un conjunto $A \subseteq N$ es r.e. sii una función P-computable $f_A : N \rightarrow N$ definida por :

$f_A(x) =$ 1 si $x \in A$
indefinida en caso contrario

Dem.-
$\Rightarrow$) De acuerdo al teorema 15, existe una función ϕ_q tal que $A = dom(\phi_q)$. Entonces el programa :

```
PROGRAM(X0)
     X1 := EVAL-PROG (q,X0);
     X2 := 1
RESULT(X2)
```

computa la función f_A

$\Leftarrow$) Inmediato de acuerdo al teorema 15.

CONJUNTOS DECIDIBLES

Def.- Un conjunto A es decidible si su función característica C_A es computable.

$C_A(x) =$ 1 si $x \in A$
0 en caso contrario

Teorema 17.- Si A y B son decidibles, también lo son :
C(A) (complemento de A)
$A \cup B$
$A \cap B$

Dem.- Es inmediato construir los programas que computan las funciones características en cada caso.

Teorema 18.- Todo conjunto decidible es r.e..

Dem.- Sea A decidible, q el índice del programa que computa C_A
Considere el siguiente programa :

```
PROGRAM(X0)
      X1 := EVAL-PROG(q,X0);
      WHILE X1 = 0 DO SKIP END
RESULT(X1)
```

la función que computa vale 1 si $x \in A$ y es indefinida en caso contrario. Por teorema 16, A es r.e..

Sea $K = \{ i \in N / <Ix(i),i> \downarrow \}$

Teorema 19 - K es r.e. pero no es decidible

Dem.-
1) K no es decidible

$$C_K(i) = \begin{cases} 1 & \text{si } <Ix(i),i> \downarrow \\ 0 & \text{en caso contrario} \end{cases}$$

es la función θ , que sabemos que no es computable

2) K es r.e.
Consideremos el siguiente programa :

```
PROGRAM(X0)
      X1 := EVAL-PROG(XO,XO);
      X1 := 1
RESULT(X1)
```

La función que evalúa vale 1 si $x \in K$ y es indefinida en caso contrario. Por teorema 16, K es r.e.

Existe una interesante relación entre conjuntos decidibles y conjuntos r.e., que se establece en el teorema siguiente :

Teorema 20 .- Si A y C(A) (complemento de A) son ambos r.e., entonces ambos son decidibles.

Dem.- Si $A = \Phi$ o $A = N$, inmediato

Si no, sean MA y MCA expresiones que generan A y C(A).

El programa siguiente computa la función característica de A.

```
PROGRAM(X0)
     X1 := 0;
     WHILE MA(X1) /= X0 & MCA(X1) /= X0 DO
          X1 := SUC(X1)
     END;
     IF MA(X1) = X0
          THEN X2 := 1
          ELSE X2 := 0
     FI
RESULT(X2)
```

Un programa análogo (cambiando la sentencia IF) computa la función característica de C(A).

Como consecuencia de este teorema y el teorema 19 ,tenemos que el conjunto:

$C(K) = \{ i \in N / \langle Ix(i),i \rangle \uparrow \}$ no es r.e.

Def.- Un **conjunto I de índices** (códigos de programas en P) **respeta funciones** si $i \in I$ y $\phi_i = \phi_j$ implica $j \in I$.

Ejemplos :

$I1 = \{ i \in N / \langle Ix(i),n \rangle \downarrow$ en a lo sumo m pasos $\}$ no respeta funciones

$I2 = \{ i \in N / \phi_i (x) = x + 1 \}$ respeta funciones

Teorema 21.- (Teorema de Rice). Sea I un conjunto de índices que respeta funciones , $I \neq \emptyset$, $I \neq N$. I es indecidible.

Dem.- Sea I que respeta funciones, $q \in I$. Supongamos que ningún índice para la función vacía pertenece a I. (Esto no nos hace perder generalidad, si los índices de la función vacía pertenecen a I tomamos C(I)).

Sea MH la macro en P que dado n computa el código h(n) del siguiente programa H(n) :

```
PROGRAM(X0)
     X1 := EVAL-PROG(n,n);
     X1 := EVAL-PROG(q,X0)
RESULT(X1)
```

Notar que $h(n) \in I$ sii $\langle Ix(n),n \rangle \downarrow$

Supongamos que I es decidible, sea MI una macro que computa C_I.

El siguiente programa :

```
PROGRAM(X0)
     X1 := MH(X0);
     X1 := MI(X1)
RESULT(X1)
```

computa la función θ !!

Bibliografía

1. Notas del curso *Computability*, Prof. Brör Bjerner

2. *Computability and Logic*, Daniel Cohen

■