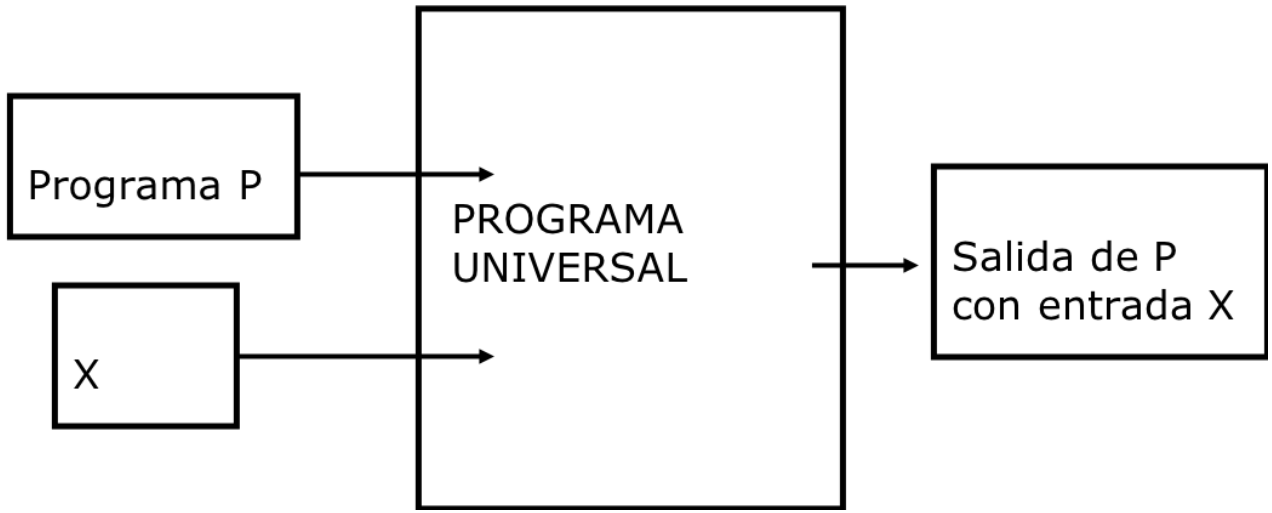
Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

Resultados Fundamentales y Resolubilidad de Problemas

Departamento de Informática
Universidad de Oviedo

El programa Universal

- En este tema, las funciones computables (programas) serán la entrada de otras funciones computables (programas)
- El **programa universal** toma como entrada un programa **P**, y una entrada **X** para él, y devuelve el resultado de ejecutar **P** con entrada **X**.



Contenido

Parte I

Enumeración de los algoritmos

La idea de la codificación

- Codificar un programa mediante un número natural nos permitiría utilizar programas como entrada de otros programas
- La idea para codificar es muy simple: asignamos un número único a cada programa
- La arquitectura von Neumann está basada en algo similar, ya que los programas y los datos se almacenan de la misma forma
- El primero en utilizar una codificación de un sistema formal fue Kurt Gödel en la demostración de sus famosos Teoremas de Incompletitud

Nuestra codificación

- Hay muchas formas (equivalentes computacionalmente) de codificar programas
- Una de las más simples consiste en imitar la forma en la que codificamos programas en nuestros ordenadores:
  - Primero, asignamos un número de 8 bits a cada posible símbolo (su código ASCII)
  - Después, concatenamos los números de todos los simbolos del programa
- El resultado será un número natural (que normalmente será enorme) que será único para cada programa y al que llamaremos el **código** del programa

UN EJEMPLO

- Considera el siguiente programa

```
begin
    X1:=0
end
```

- Su código en binario sería

01100010	**01100101**	**01100111**	**01101001**
01101110	**00001010**	**00100000**	**00100000**
01011000	**00110001**	**00111010**	**00111101**
00110000	**00001010**	**01100101**	**01101110**
01100100			

- Que corresponde al número natural
  334824609307739149582812355963437043830 76

Propiedades de la codificación (I)

- Si extendemos el modelo de los programas while para incluir variables y operaciones de manipulación de caracteres, la codificación descrita es claramente computable
- Por tanto, la función

$$\textbf{cod} : \textbf{Programas} \to \mathbb{N}$$

  es computable
- **cod** es también total (todo programa tiene un código) e inyectiva (programas diferentes tienen códigos diferentes)
- Además, dado un número que codifica un programa, podemos obtener de nuevo dicho programa
- Sin embargo, hay números que no son el código de ningún programa (por ejemplo, 0)

DEFINIENDO *decod*

- Definimos la función $decod : \mathbb{N} \to Programas$ como sigue

$$decod(n) = \begin{cases} P & \text{si existe } P \text{ tal que } cod(P) = n \\ Q & \text{en otro caso} \end{cases}$$

donde Q es un programa fijo, por ejemplo

```
begin
    X1:=0
end
```

Propiedades de la codificación (II)

- Obsérvese que *decod* puede devolver Q en infinitos casos (uno para cada número que no codifique ningún programa)
- Esto no es importante porque:
  - Sólo estamos interesados en las funciones computadas por los programas
  - Cada función computable es computada por infinitos programas diferentes (la función semántica de un programa no cambia si añadimos una o más veces una instrucción como $X2 := 0$ al final)
- Es fácil comprobar que $decod(cod(P)) = P$
- También se puede ver que *decod* es total (trivial) y computable (se podría comprobar si un número codifica un programa while correcto utilizando expresiones regulares y gramáticas libres de contexto)

En resumen

- Hemos definido funciones

$$cod : Programas \to \mathbb{N}$$

y

$$decod : \mathbb{N} \to Programas$$

- Ambas son totales y computables
- *cod* es además inyectiva
- Se cumple $decod(cod(P)) = P$
- *Pregunta: ¿Se cumple* $cod(decod(n)) = n$*?*

Algo de notación

- Recuerda que a cada programa P le corresponden infinitas funciones computables, una por cada posible aridad:

$$\varphi_P^{(j)} : \mathbb{N}^j \to \mathbb{N}$$

- En lugar de P, utilizaremos a menudo el código de P como subíndice en la expresión anterior. Es decir:

$$\varphi_e^{(j)} = \varphi_P^{(j)}$$

si $e = cod(P)$

Parte II

Teoremas fundamentales

La idea de una Función Universal

- Ahora que ya podemos codificar programas mediante números, podemos tratar las entradas de una función como si fuesen programas
- Por ejemplo, considérese la función

$$\Phi(e, x) = \varphi_e(x)$$

- Esta función, siendo e el código de un programa, y x un número, devolverá el resultado de ejecutar el programa $P = decod(e)$ con entrada x

La idea de una Función Universal

- Decimos que esta función es **universal** porque podemos utilizarla para reproducir el comportamiento de *c*ualquier función computable unaria ya que
  - **Universal en cuanto a programas**: El primer argumento **e** de la función de Universalidad $\Phi(e, x)$ es la codificación de un programa. Al recorrer dicho argumento todos los naturales, se alcanzan todos los programas que existen.
  - **Universal en cuanto a entradas**: El resto de los argumentos son las entradas para el programa P_e que se debe "simular" con $\Phi(e, x)$, fijada de antemano la aridad j de su función semántica.
- Es más, podemos considerar una función universal para cada aridad:

$$\Phi(e, x_1, x_2, \ldots, x_j) = \varphi_e^{(j)}(x_1, x_2, \ldots, x_j)$$

La Función Universal es computable

- Aunque al principio parece casi increible, es posible probar que cada función universal $\Phi(e, x_1, x_2, \ldots, x_j)$ es computable
- Podemos proceder como sigue:
  - Primero, obtenemos $P = decod(e)$ (ya sabemos que eso es computable)
  - Después, simulamos la ejecución de P con entrada $x_1, x_2, \ldots, x_j$
  - Para acabar, devolvemos el resultado de la computación anterior
- El segundo paso requiere una demostración larga y tediosa, pero se puede probar que siempre es computable
- Es comparable al funcionamiento de un intérprete o un sistema operativo

El Teorema de Universalidad

- La conclusión de la diapositiva anterior es tan importante que debemos ponerla en forma de teorema

Teorema de Universalidad

Para cada $j \geq 1$, la función universal

$$\begin{array}{rccc} \Phi: & \mathbb{N}^{j+1} & \to & \mathbb{N} \\ & \Phi^{(j+1)}(e, x_1, x_2, \ldots, x_j) & = & \varphi_e^{(j)}(x_1, x_2, \ldots, x_j) \\ & \forall e, x_1, x_2, \ldots, x_j & & \end{array}$$

es computable

Universalidad: Ejercicio (I)

- El Teorema de Universalidad nos dice que la función universal Φ^{j+1} nos permite simular todas las funciones computables de aridad j. ¿Podríamos afirmar que en realidad Φ^{j+1} es capaz de simular **todas** las funciones computables de aridad $j \leq j$?

La Macro de Universalidad

- Ya que las funciones universales son computables para cualquier aridad, hay programas while que las computan y los podemos utilizar como macros
- A partir de ahora, en nuestros programas while podremos utilizar macro-instrucciones como

$$Z := U(X, Y)$$

- Cuando esta macro se ejecuta, el valor $\varphi_X(Y)$ (es decir, el resultado de ejecutar el programa $P = decod(X)$ con entrada Y) se almacenará en Z
- Obsérvese que $\varphi_X(Y)$ podría ser indeterminado, en cuyo caso el programa se quedará permanentemente atascado en la ejecución de la macro

Universalidad: Ejercicios (II)

- Construir un programa while P que calcule la siguiente función ternaria. Se pueden emplear las macros de asignación, producto y de Universalidad.

$$f(x, y, z) = \varphi_x(z) * \varphi_y(z)$$

- Construir un programa while P que calcule la siguiente función ternaria. Se pueden emplear las macros de asignación y de Universalidad.

$$f(x, y, z) = \varphi_x(\varphi_y(z))$$

UNIVERSALIDAD: SOLUCIÓN DE EJERCICIOS (II)

- Construir un programa while P que calcule la siguiente función ternaria. Se pueden emplear las macros de asignación, producto y de Universalidad.

$$f(x, y, z) = \varphi_x(z) * \varphi_y(z)$$

```
begin
    X1:=U(X1,X3);
    X2:=U(X2,X3);
    X1:=X1*X2
end
```

Universalidad: Solución de Ejercicios (II)

- Construir un programa while P que calcule la siguiente función ternaria. Se pueden emplear las macros de asignación, producto y de Universalidad.

$$f(x, y, z) = \varphi_x(\varphi_y(z))$$

```
begin
    X2:=U(X2,X3);
    X1:=U(X1,X2)
end
```

La idea de la Parametrización

- A veces, cuando tenemos una función, es útil fijar algunas de sus entradas para obtener otras funciones
- Por ejemplo, si tenemos

$$f(x, y) = x * y$$

y fijamos $x = 2$ obtendremos una nueva función, con una única variable, a la que podemos llamar g:

$$g(y) = f(2, y) = 2 * y$$

- Ocurre algo similar en algunos lenguajes de programación, como C++ o Python, cuando definimos funciones con parámetros por defecto

Parametrización y Programas (I)

- Supón que tenemos un programa P que computa $f(x, y) = x * y$. Ahora considera el programa Q

```
begin
    X2:=X1;
    X1:=2;
    P
end
```

- Claramente, la función unaria semántica de Q es

$$g(y) = f(2, y) = 2 * y$$

y si conocemos $cod(P)$ podríamos computar facilmente $cod(Q)$

Parametrización y Programas (II)

- Podemos generalizar el ejemplo anterior. Considérese el siguiente programa:

```
begin
    X2:=X1;
    X1:=C;
    P
end
```

  donde C es una constante.
- Entonces, la función unaria semántica de este nuevo programa es $h(y) = f(C, y) = C * y$
- Más importante, **podemos computar** el código de este nuevo programa a partir de $cod(P)$ y del valor C

Parametrización y Programas (III)

- El programa P de los ejemplos no tiene nada de particular, así que podemos utilizar *cualquier* programa P
- Cuando variamos P, estamos considerando todas las funciones computables binarias $\varphi_P(x, y)$ y obtenemos las funciones computables unarias

$$g_{(P,C)}(y) = \varphi_P(C, y)$$

- Nótese que reducimos (fijamos) un parámetro y la nueva función depende de P y de C
- De hecho, no sólo podemos considerar funciones de dos variables, sino de todas las variables que queramos

Parametrización en general

- Considera un programa P, un número de variables m que se fijan, y un número de variables n que seguirán libres
- Considera también m constantes $C_1, \ldots, C_m$ y el programa

```
begin
    Xm+1:=X1;
    ...
    Xm+n:=Xn;
    X1:=C1;
    ...
    Xm:=Cm;
    P
end
```

- La función semántica n-aria de este programa es $h(x_1, \ldots, x_n) = \varphi_P^{(m+n)}(C_1, \ldots, C_m, x_1, \ldots, x_n)$
- Y, de nuevo, **podemos computar** el código de este nuevo programa a partir de $cod(P)$ y de los valores $C_1, \ldots, C_m$

El Teorema de Parametrización

- Todo el razonamiento anterior nos lleva al Teorema de Parametrización (también conocido como el Teorema *s-m-n*)

Teorema de Parametrización

Para cada $m \geq 1$ y $n \geq 1$ existe una función total y computable s_n^m tal que

$$\varphi^{(n)}_{s_n^m(e,y_1,\ldots,y_m)}(x_1,\ldots,x_n) = \varphi_e^{(m+n)}(y_1,\ldots,y_m,x_1,\ldots,x_n)$$

para todo $e, y_1, \ldots, y_m, x_1, \ldots, x_n$

Parametrización: Ejercicios (I)

1. Consideremos e tal que $\varphi_e(x, y) = f(x, y) = 3x - 2y$. Aplicando el teorema de parametrización, ¿Cuál sería el número a que codifica el programa while con función semántica unaria $\varphi_a(y) = g(y) = 6 - 2y$?

2. Dadas las funciones unarias $f_1(x)$ y $f_2(x)$, computadas respectivamente por programas de código e_1 y e_2; demuéstrese que la función unaria $f(x) = f_1(x) * f_2(x)$ es computable para cualesquiera que sean las funciones f_1 y f_2, y que el código del programa que calcula f existe siempre y depende de e_1 y e_2.

Parametrización: Solución de Ejercicio (I)

1. Consideremos e tal que $\varphi_e(x, y) = f(x, y) = 3x - 2y$. Aplicando el teorema de parametrización, ¿Cuál sería el número a que codifica el programa while con función semántica unaria $\varphi_a(y) = g(y) = 6 - 2y$?

- El programa $decod(a)$ debería invocar a $\varphi_e(x, y)$ con $x = 2$, y así devolvería $6 - 2y$.
- Parametriza x, luego su código es: $a = s_1^1(e, 2)$

Parametrización: Solución de Ejercicio (I)

2. Dadas las funciones unarias $f_1(x)$ y $f_2(x)$, computadas respectivamente por programas de código e_1 y e_2; demuéstrese que la función unaria $f(x) = f_1(x) * f_2(x)$ es computable para cualesquiera que sean las funciones f_1 y f_2, y que el código del programa que calcula f existe siempre y depende de e_1 y e_2.

Reescribimos el enunciado: Sean $f_1(X) = \varphi_{e_1}(x)$ y $f_2(X) = \varphi_{e_2}(x)$

- Demostrar que $f(x) = \varphi_{e_1}(x) * \varphi_{e_2}(x)$ es computable $\forall e_1, e_2$
- Si $f(x) = \varphi_e(x)$, e existe $\forall e_1, e_2$ y depende de ellos.

Parametrización: Solución de Ejercicio (2) (I)

Demostrar que $f(x) = \varphi_{e_1}(x) * \varphi_{e_2}(x)$ es computable $\forall e_1, e_2$

- Podríamos facilmente computar f como sigue:

begin
$X2 := U(e_1, X1)$;
$X1 := U(e_2, X1)$;
$X1 := X1 * X2$
end

Parametrización: Solución de Ejercicio (3) (I)

Si $f(x) = \varphi_e(x)$, e existe $\forall e_1, e_2$ y depende de ellos.

- Para probar que esto sirve para cualquier valor de e_1 y e_2 podríamos ver este programa como un caso particular de:

$$\begin{array}{l}\text{begin}\\ \quad X1 := U(X1, X3);\\ \quad X2 := U(X2, X3);\\ \quad X1 := X1 * X2\\ \text{end}\end{array}$$

- Si a es el código de este programa:

$$\varphi_a(x, y, z) = \varphi_x(z) * \varphi_y(z)$$

Parametrización: Solución de Ejercicio (4) (I)

- $\ldots a$ es $\ldots$: $\varphi_a(x, y, z) = \varphi_x(z) * \varphi_y(z)$
- Por el teorema de parametrización, existe una función s_1^2 **total y computable**:

$$\varphi_{s_1^2(a,x,y)}(z) = \varphi_a(x, y, z), \forall x, y, z$$

- para cualesquiera e_1 y e_2 podemos computar $e = s_1^2(a, e_1, e_2)$ tal que:

$$\varphi_e(x) = f(x) = \varphi_{e_1}(x) * \varphi_{e_2}(x)$$

- luego f es computable $\forall e_1, e_2$, su código es e, que existe $\forall e_1, e_2$ y depende de ellos.

PARAMETRIZACIÓN: EJERCICIOS (II)

3. Dado el siguiente programa P:

```
begin
    X1:=U(e,k)
    X1:=2
end
```

   - Determinar su función semántica unaria , $\varphi_P(x)$
   - ¿es una función constante?, ¿de qué depende?
   - Demostrar que para cualquier e y cualquier k, se puede obtener el código, c, de un programa tal que $\varphi_c(x) = \varphi_P(x)$, y que c depende de e y k.

PARAMETRIZACIÓN: SOLUCIÓN EJERCICIO (II)

Dado el siguiente programa P:

```
begin
    X1:=U(e,k)
    X1:=2
end
```

- Determinar su función semántica unaria , $\varphi_P(x)$

$$\varphi_P(x) = \begin{cases} 2 & \text{si } P_e \text{ para con entrada } k \\ \perp & \text{en otro caso} \end{cases}$$

- No es una función constante, ya que una función indefinida no es nunca constante. Depende de e y de k.

Parametrización: Solución Ejercicio (2) (II)

Demostrar que para cualquier e y cualquier k, se puede obtener el código, c, de un programa tal que $\varphi_c(x) = \varphi_P(x)$, y que c depende de e y k.

- Podemos considerar que P es un caso particular del programa:

```
begin
   X1:=U(X1,X2)
   X1:=2
end
```

- Este es un programa e' tal que:

$$\varphi_{e'}(x, y, z) = \begin{cases} 2 & \text{si } P_x \text{ para con entrada } y \\ \perp & \text{en otro caso} \end{cases}$$

Parametrización: Solución Ejercicio (3) (II)

- Por el teorema de parametrización, existe una función s_1^2 **total y computable**:

$$\varphi_{s_1^2(e',x,y)}(z) = \varphi_{e'}(x, y, z), \forall x, y, z$$

- Así:

$$\varphi_{s_1^2(e',e,k)}(x) = \varphi_{e'}(e, k, x) = \varphi_P(x)$$

- $c = s_1^2(e', e, k)$

Parametrización: Solución Ejercicio (4) (II)

- Si quisiéramos demostrar que $\varphi_c(x) = \varphi_P(x)$, y que c depende sólo de e y k deberíamos dar un paso más
- Como s_1^2 es computable, existe a tal que:

$$\varphi_a(x, y, z) = s_1^2(x, y, z) = \forall x, y, z$$

- Aplicando de nuevo parametrización, existe s' total y computable tal que:

$$\varphi_{s_2'^1(a,x)}(y, z) = \varphi_a(x, y, z)$$

- lo que significa que hay una función computable, que depende solo de y y z que computa el código del programa anterior.

$$\varphi_{s_2'^1(a,e')}(e, k) = \varphi_a(e', e, k) = s_1^2(e', e, k) = c$$

Parametrización: Ejercicios (III)

4. Demostrar que para cualquier e (número que codifica un programa) y cualquier k (número que codifica unos datos de entrada para él) se puede obtener el código de un programa que, dado x, devuelve $x + 1$ si el programa P_e al ejecutarse con input k para.
   - ¿Qué hace el programa cuyo código has encontrado si el programa P_e al ejecutarse con input k no para?

Parametrización: Solución Ejercicio (III)

- Tal programa podría ser un caso particular del siguiente (de código e'):

```
begin
   X1:=U(X1,X2);
   X1:=succ(X3)
end
```

$$\varphi_{e'}(x,y,z) = \begin{cases} z+1 & \text{si } P_x \text{ para con entrada } y \\ \perp & \text{en otro caso} \end{cases}$$

- donde $x = e$ e $y = k$.

Parametrización: Solución Ejercicio (III)

- Por el teorema de parametrización, existe una función total y computable s_1^2 tal que:

$$\varphi_{s_1^2(e',e,k)}(x) = \varphi_{e'}(e,k,x)$$

$$\varphi_{s_1^2(e',e,k)}(x) = \begin{cases} x+1 & \text{si } P_e \text{ para con entrada } k \\ \perp & \text{en otro caso} \end{cases}$$

- $a = s_1^2(e',e,k)$ es el código del programa buscado.
- si P_e no para con entrada k, el programa de código a tampoco para.

El Teorema de Recursión

- El tercer resultado principal de este tema es una consecuencia útil y poderosa de los Teoremas de Universalidad y de Parametrización

Teorema de Recursión

Si f es una función total y computable, entonces existe un número natural e tal que

$$\varphi_{f(e)} = \varphi_e$$

Demostración del Teorema de Recursión (I)

- Considérese una función total y computable f
- Podemos escribir el siguiente programa $P_{(C)}$, que depende de un parámetro C

```
begin
    X2:=U(C,C);
    X1:=U(X2,X1);
end
```

- Por el teorema de parametrización, la función definida por $h(C) = cod(P_{(C)})$ es total y computable
- Nótese que, si φ_C está definida en C, entonces

$$\varphi_{h(C)} = \varphi_{\varphi_C(C)}$$

Demostración del Teorema de Recursión (II)

- Ahora, considérese $g = f \circ h$ (esto es, $g(x) = f(h(x))$)
- Obviamente, g es total y computable. Por tanto:
  - por ser computable, podemos considerar a tal que $g = \varphi_a$.
  - por ser total $g = \varphi_a$ está definida en a.
- Como sabemos que $\varphi_{h(C)} = \varphi_{\varphi_C(C)}$ cuando $\varphi_C(C)$ está definido, tenemos que

$$\varphi_{h(a)} = \varphi_{\varphi_a(a)} = \varphi_{g(a)} = \varphi_{f(h(a))}$$

- Se puede tomar $e = h(a)$ como el número e en el enunciado del Teorema de Recursión.

Aplicando el Teorema de Recursión (I)

- Una de las principales aplicaciones del Teorema de Recursión es la posibilidad de definir funciones computables mediante recursión (¡era de esperar!)
- Por ejemplo, veamos cómo demostrar que la función factorial es computable utilizando el Teorema de Recursión
- Consideremos el programa

```
begin
    if X2=0
        X1:=1
    else
        X3:=pred(X2)
        X1:=X2*U(X1,X3)
end
```

Aplicando el Teorema de Recursión (II)

- El programa de la diapositiva anterior tiene la siguiente función binaria semántica

$$g(x,y) = \begin{cases} 1 & \text{si } y = 0 \\ y * \varphi_x(y-1) & \text{en otro caso} \end{cases}$$

- El Teorema de Parametrización nos permite deducir que existe una función total y computable f tal que

$$\varphi_{f(x)}(y) = g(x,y)$$

Aplicando el Teorema de Recursión (III)

- Ahora, utilizamos el Teorema de Recursión para obtener e tal que $\varphi_{f(e)} = \varphi_e$ y entonces tenemos

$$\varphi_e(y) = \varphi_{f(e)}(y) = g(e, y) = \begin{cases} 1 & \text{si } y = 0 \\ y * \varphi_e(y - 1) & \text{en otro caso} \end{cases}$$

y, por lo tanto, φ_e es la función factorial.

Recursión: Ejercicios

1. Demuestra, utilizando el teorema de recursión, que **existe** un número n tal que

$$\varphi_n(x, y) = \varphi_x(n * y), \forall x, y \in \mathbb{N}$$

2. Demuestra, utilizando el teorema de recursión, que **existe** un número n tal que

$$\varphi_n(x, y, z) = \varphi_z(\varphi_y(n)) * \varphi_x(n), \forall x, y, z \in \mathbb{N}$$

Recursión: Resolución Ejercicio

1. Demuestra que:

$$\exists n : \varphi_n(x, y) = \varphi_x(n * y), \forall x, y \in \mathbb{N}$$

- Es posible construir un programa P tal que:

$$\varphi_e(z, x, y) = \varphi_x(z * y), \text{donde } e = cod(P)$$

- por ejemplo:

```
begin
    X3:=X1*X3;
    X1:=U(X2,X3)
end
```

- por el T. de parametrización, existe s_2^1 total y computable:

$$\varphi_{s_2^1(e,z)}(x, y) = \varphi_e(z, x, y) = \varphi_x(z * y)$$

Recursión: Resolución Ejercicio (2)

- Como $s_2^1(e, z)$ es computable, podemos encontrar un programa de código e' tal que:

$$\varphi_{e'}(e, z) = s_2^1(e, z)$$

- aplicando de nuevo el T. de parametrización, existe otra función total y computable:

$$\varphi_{s_1'^1(e',e)}(z) = \varphi_{e'}(e, z) = s_2^1(e, z)$$

- Llamemos $f(z)$ a $\varphi_{s_1'^1(e',e)}(z)$, así:

$$\varphi_{f(z)}(x, y) = \varphi_e(z, x, y) = \varphi_x(z * y)$$

- Aplicando el T. de recursión a f tenemos que existe un n:

$$\varphi_n(x, y) = \varphi_{f(n)}(x, y) = \varphi_x(n * y)$$

Recursión: Resolución Ejercicio (3)

2. Demuestra que:

$$\exists n : \varphi_n(x, y, z) = \varphi_z(\varphi_y(n)) * \varphi_x(n), \forall x, y, z \in \mathbb{N}$$

- Es posible construir un programa P tal que:

$$\varphi_e(v, x, y, z) = \varphi_z(\varphi_y(v)) * \varphi_x(v), \text{donde } e = cod(P)$$

- por ejemplo:

```
begin
    X3:=U(X3,X1);
    X3:=U(X4,X3);
    X1:=U(X2,X1);
    X1:=X1*X3;
end
```

- por el T. de parametrización, existe s_3^1 total y computable:

$$\varphi_{s_3^1(e,v)}(x, y, z) = \varphi_e(v, x, y, z) = \varphi_z(\varphi_y(v)) * \varphi_x(v)$$

Recursión: Resolución Ejercicio (4)

- Como $s_3^1(e, v)$ es computable, podemos encontrar un programa de código e' tal que:

$$\varphi_{e'}(e, v) = s_3^1(e, v)$$

- aplicando de nuevo el T. de parametrización, existe otra función total y computable:

$$\varphi_{s_1'^1(e',e)}(v) = \varphi_{e'}(e, v) = s_3^1(e, v)$$

- Llamemos $f(v)$ a $\varphi_{s_1'^1(e',e)}(v)$, así:

$$\varphi_{f(v)}(x, y, z) = \varphi_e(v, x, y, z) = \varphi_z(\varphi_y(v)) * \varphi_x(v)$$

- Aplicando el T. de recursión a f tenemos que existe un n:

$$\varphi_n(x, y, z) = \varphi_{f(n)}(x, y, z) = \varphi_z(\varphi_y(n)) * \varphi_x(n)$$

Divirtiéndonos con el Teorema de Recursión: Quines (I)

- Un **quine** es un programa que ignora la entrada y simplemente devuelve su propio código
- El término fue acuñado por Douglas Hofstadter, en honor del filósofo Willard Van Orman Quine, en su extraordinario libro *Gödel, Escher, Bach*
- Un quine en python

```
s = 's = %r\nprint(s%%s)'
print(s%s)
```

- Un quine en Matlab/Octave

```
s='disp(char([115,61,39,s,39,59,s]));';disp(char([115,61,39,s,39,59,s]));
```

Divirtiéndonos con el Teorema de Recursión: Quines (II)

- Podemos construir fácilmente funciones computables que son quines (esto es, que siempre devuelven su propio código) con la ayuda del Teorema de Recursión
- Considera la función computable

$$g(x, y) = x$$

- Por el Teorema de Parametrización, existe f total y computable tal que

$$\varphi_{f(x)}(y) = g(x, y) = x$$

- Si aplicamos el Teorema de Recursión a f obtenemos e tal que $\varphi_{f(e)} = \varphi_e$ y entonces

$$\varphi_e(y) = \varphi_{f(e)}(y) = g(e, y) = e$$

- Esto ocurre para todo y, luego φ_e es un quine

Parte III

Resolubilidad e Irresolibilidad Algorítmicas

Un poco de Historia

- En 1928, David Hilbert y Wilhelm Ackerman propusieron el Entscheidungsproblem (“problema de la decisión”, en alemán) para la lógica de primer orden

Entscheidungsproblem

Encuéntrese un **algoritmo** para determinar si una **fórmula de lógica de primer orden dada** es válida o no.

- Para nosotros, un algoritmo como ese sería extremadamente útil

¿Qué pedían Hilbert y Ackerman?

- Tenemos un algoritmo (resolución general) que responde “sí” siempre que la fórmula dada es válida y, en algunas ocasiones, también responde “no” cuando la fórmula no es válida
- Hilbert y Ackerman querían un algoritmo que respondiera correctamente para **cualquier** fórmula, no solamente en casos particulares.
- El problema se podría resolver de dos maneras distintas:
  - Encontrando un algoritmo y demostrando que resuelve correctamente todos los casos
  - Demostrando que **dicho algoritmo no existe**
- Finalmente, se demostró que no existe un método algorítmico para resolver el problema, pero... ¿**cómo podríamos demostrar que un algoritmo no existe**?

Demostrar: no existe un algoritmo para un problema

- Para demostrar que no existe un algoritmo para resolver un determinado problema, típicamente se procede de la siguiente manera:
  - Fijamos un modelo de computación (p.e. Programas While).
  - Demostramos que el problema no se puede resolver empleando dicho modelo, utilizando técnicas como:
    - Demostración por contradicción (¿Recuerdas?)
    - Diagonalización
    - Reducción
    - ...
- Entonces, invocamos la **tesis de Church-Turing** (o simplemente consideramos que todos los modelos de computación conocidos son equivalentes)
- En esta situación, decimos que el problema es **irresoluble** o **indecidible** (algorítmicamente)

Problemas de decisión

- Hemos dicho que para demostrar que un problema es irresoluble, trabajamos con un modelo de computación
- Pero los modelos de computación se refieren a **funciones matemáticas**, no a problemas
- Relacionamos ambos conceptos mediante lo que denominamos **problemas de decisión**

Problema de decisión

Un **problema de decisión** es una pregunta con respuesta sí/no que depende de algunos parámetros de entrada

Ejemplos de problemas de decisión

En Informática, Lógica y Matemáticas hay muchos ejemplos de problemas de decisión

- Decidir si un número natural dado es primo o no
- Decidir si una fórmula dada de lógica proposicional es una tautología o no
- Decidir si una fórmula dada de lógica de predicados es una tautología o no
- Decidir si un programa dado es correcto sintácticamente o no
- Decidir si una gramática libre de contexto dada es ambigua o no
- Decidir si un programa dado para (termina) o no
- Decidir si un programa dado es un virus o no

Problemas de decisión y funciones computables

- Podemos transformar problemas de decisión generales en funciones sobre enteros utilizando **codificación**
- Si codificamos ecuaciones, fórmulas, gramáticas, programas... mediante números naturales, entonces todos esos problemas se pueden ver como funciones sobre los números naturales que devuelven 0 (falso) o 1 (verdadero)
- La función asociada a un problema de decisión se denomina su **función característica**

Computabilidad y resolubilidad

Demostrar que un problema de decisión es resoluble (resp. irresoluble) es equivalente a demostrar que su función característica es computable (resp. no computable)

El problema de la parada

- El problema de la parada es la piedra angular en el estudio de la irresolubilidad: es un problema irresoluble y puede utilizarse para demostrar que muchos otros problemas son irresolubles
- Se puede formular para cualquier modelo de computación, pero aquí nos centraremos en su versión para los Programas While

El problema de la parada para Progamas While

Decidir si un Programa While dado devuelve un valor (para) con una entrada dada.

El problema de la parada es irresoluble

- Para demostrar que el problema de la parada es irresoluble adaptamos la demostración clásica de Turing (1936)
- Es un ejemplo de demostracción por contradicción
- También está profundamente ligada a la paradoja del mentiroso ("*Estoy mintiendo*") y a las demostraciones de Gödel de sus teoremas de incompletitud
- Puedes ver una versión animada de la demostración en el siguiente vídeo (en inglés): https://www.youtube.com/watch?v=92WHN-pAFCs
- O, si prefieres la poesía, puedes leer la demostración en verso (en inglés): http://www.lel.ed.ac.uk/ gpullum/loopsnoop.html

La demostración de irresolubilidad (I)

- Empezamos suponiendo que existe una macro $H(p, a)$ que decide el problema de la parada para Programas While
- Es decir:
  - Si el programa codificado por p para con la entrada a entonces $H(p, a)$ devuelve 1
  - Si no, $H(p, a)$ devuelve 0
- A partir de este supuesto, vamos a llegar a una contradicción

La demostración de irresolubilidad (II)

- Ahora, podemos construir el siguiente programa

```
begin
    X2 := H(X1,X1);
    if X2 = 0 then
       X1:= 0;
    else
        while X1 = X1 do
            X1:=succ(X1);
end
```

- Dado que *H* es una macro, este es un programa While válido y, por lo tanto, tendrá un código *c*

La demostración de irresolubilidad (III)

- Ahora, piensa qué pasaría si la entrada del programa fuera su propio código c
- ¿Pararía o no?
- Podemos distinguir dos casos:
  - Si $H(c, c) = 0$, entonces el programa pararía y retornaría 0. Pero entonces $H(c, c)$ debería haber sido 1!!!!!
  - Si $H(c, c) = 1$, entonces el programa entraría en un bucle infinito. Pero entonces $H(c, c)$ debería haber sido 0!!!!!
- Hemos alcanzado una contradicción en los dos casos, por lo que es imposible que la macro H exista
- Concluimos que el problema de la parada es irresoluble

El método de Reducción

- Ahora que sabemos que un problema concreto es irresoluble, muchos otros caerán también, como piezas de un dominó...
- Para ello, emplearemos el método de **reducción**

Reducción

Decimos que un problema de decisión P se reduce a otro problema de decisión P' si a partir de un algoritmo que resuelva P' podemos **construir** un algoritmo que resuelva P.

Uso del método de reducción

- Nótese que si podemos reducir P a P', significa que P es *más fácil* de resolver que P'. Esto se denota por $P \leq P'$.
- Nótese también que si P se puede reducir a P' pero sabemos que P es irresoluble, entonces P' (que es más difícil) debe ser necesariamente irresoluble también
- Esto es sencillo de demostrar:
  - Sabemos que P es irresoluble y suponemos que P se puede reducir a P'
  - Suponemos además que existe un algoritmo que resuelve P'
  - Pero, entonces, dado que $P \leq P'$, a partir del algoritmo que resuelve P' podemos **construir** un algoritmo para P
  - Esto es una contradicción, ya que sabemos P es irresoluble
- Por tanto, si podemos reducir el problema de la parada a otros problemas sabremos inmediatamente que dichos otros problemas también son irresolubles

Ejemplo de reducción (I)

- Considera el siguiente problema

Nuestro problema

Dado un Programa While P, determinar si P devuelve 2 con al menos una entrada.

- Este problema es irresoluble, y lo vamos a demostrar primero mediante el método de reducción

Ejemplo de reducción (II)

- La estrategia típica es reducir el problema de la parada (que sabemos es irresoluble) a nuestro nuevo problema
- Empezamos suponiendo que el problema es resoluble, y por tanto existe un programa A cuya función semántica es:

$$\varphi_A(x) = \begin{cases} 1 & \text{si } P_x \text{ devuelve devuelve 2 con al menos una entrada} \\ 0 & \text{en otro caso} \end{cases}$$

- A partir de A, podemos crear una macro $A(X)$ para utilizar en otros programas while.

Ejemplo de reducción (III)

- Al algoritmo anterior deberá funcionar para cualquier Programa P_x, incluído el siguiente:

```
begin
    X1:=U(c,k);
    X1:=2
end
```

- siendo c el código de un programa y k su entrada
- la función semántica del programa es:

$$\varphi_d(x) = \begin{cases} 2 & \text{si } P_c \text{ para con entrada } k \\ \perp & \text{en otro caso} \end{cases}$$

Ejemplo de reducción (IV)

- Si el programa c no para con la entrada k, entonces el programa anterior no devolverá nada con ninguna entrada (c y k son constantes)
- Si el programa c para con la entrada k, entonces el programa anterior devolverá 2 con al menos una entrada (de hecho, con todas las entradas)
- Por el **Teorema de Parametrización**, sabemos que el código d del programa que calcula esta función semántica será una función total y computable $f(c, k)$ que depende de c y k.

Ejemplo de reducción (V)

- Dado que f es total y computable, podemos crear este otro programa

```
begin
    X1:=f(X1,X2);
    X1:=A(X1)
end
```

- dónde $A(X)$ es la macro definida anteriormente
- su función semántica sería:

$$\varphi(c,k) = \begin{cases} 1 & \text{si } P_d \text{ devuelve devuelve 2 con al menos una entrada} \\ 0 & \text{en otro caso} \end{cases}$$

Ejemplo de reducción (VI)

- Dado que conocemos φ_d, podemos ir un paso más lejos
- $\varphi_d(x)$ devuelve el valor 2 para al menos una entrada (de hecho para todas) si, y solo si, P_c para con entrada k
- Por tanto, la función anterior se puede reescribir como:

$$\varphi(c,k) = \begin{cases} 1 & \text{si } P_c \text{ para con entrada } k \\ 0 & \text{en otro caso} \end{cases}$$

- **Pero esa es la definición del problema de la parada!!**
- Hemos reducido el problema de la parada al nuestro, por tanto, nuestro problema es irresoluble

El problema de la totalidad (I)

- El problema de la totalidad es otro problema indecidible que está relacionado con el problema de la parada

El problema de la totalidad

Decidir si un programa While dado, devuelve un valor (para) con **todas** las entradas.

- De hecho, el problema de la totalidad es el problema de decisión sobre si la función semántica asociada a un programa es total o no

El problema de la totalidad (II)

- Utilizaremos el método de reducción para demostrar que el problema de la totalidad es irresoluble
- Igual que en el ejemplo anterior, supondremos que el problema es resoluble, y por tanto existe un programa T cuya función semántica es:

$$\varphi_T(x) = \begin{cases} 1 & \text{si } P_x \text{ para con todas las entradas} \\ 0 & \text{en otro caso} \end{cases}$$

- Y a partir de T, definimos una macro $T(X)$ para utilizar en otros programas while.

El problema de la totalidad (III)

- Dado el código c de un programa y una entrada k, podemos construir el siguiente programa:

```
begin
    X2:=U(c,k);
end
```

- Su función semántica es $\varphi_d(x) = x$ para toda entrada x, siempre que el programa P_c pare con entrada k
- Por el **Teorema de Parametrización**, sabemos que el código d del programa que calcula esta función semántica será una función total y computable $f(c, k)$ que depende de c y k.

El problema de la totalidad (IV)

- Dado que f es total y computable, podemos crear este otro programa

```
begin
    X1:=f(X1,X2);
    X1:=T(X1)
end
```

- dónde $T(X)$ es la macro definida anteriormente
- su función semántica sería:

$$\varphi(c,k) = \begin{cases} 1 & \text{si } P_d \text{ para con todas las entradas} \\ 0 & \text{en otro caso} \end{cases}$$

El problema de la totalidad (V)

- Sabemos que para toda entrada, $\varphi_d(x) = x$ si, y solo si, P_c para con la entrada k
- Podemos reescribir la función anterior como:

$$\varphi(c,k) = \begin{cases} 1 & \text{si } P_c \text{ para con entrada } k \\ 0 & \text{en otro caso} \end{cases}$$

- De nuevo, esta es la definición del **problema de la parada**
- Hemos reducido el problema de la parada al problema de la totalidad
- Por tanto, el problema de la totalidad es irresoluble

Propiedades semánticas (I)

- El método de reducción se puede aplicar para demostrar que muchos problemas de decisión sobre programas son irresolubles
- Para ver cómo se puede hacer para una clase muy extensa de problemas, necesitamos definir el concepto de propiedad semántica

Propiedades semánticas

Fíjese una aridad j. Una **propiedad semántica** (para la aridad j) de un programa es una propiedad que solo depende de la función semántica (de aridad j) calculada por dicho programa.

- Para simplicar, en lo que sigue fijaremos la aridad a 1

Propiedades semánticas (II)

- Algunos ejemplos de propiedades semánticas:
  - El programa para con todas las entradas
  - El programa devuelve 0 para al menos una entrada
  - El programa siempre devuelve el mismo valor
  - ...
- Al contrario, las siguientes **no** son propiedades semánticas:
  - El programa tiene un número par de líneas
  - El programa utiliza la variable X5
  - El programa no tiene instrucciones *pred*
  - ...

Propiedades semánticas (III)

- Para nosotros, el hecho más importante sobre propiedades semánticas de programas es el siguiente:

Propiedades semánticas y programas

Si S es una propiedad semántica y P_1 y P_2 son programas que tienen la misma función semántica, entonces o bien S se cumple para tanto P_1 como P_2 o no se cumple para ninguno.

- También diremos que una propiedad semántica es **no trivial** si se cumple para al menos un programa y no se cumple para al menos otro

Ejemplo sobre propiedades semánticas (I)

- Veamos cómo demostrar que una propiedad no es semántica

Ejercicio

Demuestra que la siguiente propiedad no es una propiedad semántica: El programa tiene un número par de líneas.

- Basándonos en lo que acabamos de ver, podemos afirmar que si hay un programa P_1 que cumple la propiedad y otro programa P_2 con la misma función semántica que P_1 que no la cumple, entonces no es una propiedad semántica.

Ejemplo sobre propiedades semánticas (II)

- Este programa P_1 tiene un número par de líneas:

```
begin
    X2:=0;
    X1:=0;
end
```

- Este programa P_2 **no** tiene un número par de líneas:

```
begin
    X1:=0;
end
```

- Sin embargo ambos computan la misma función semántica $\varphi(x) = 0$
- Por tanto, "tener un número par de líneas" **no** es una propiedad semántica

Teorema de Rice

- El teorema de Rice permite demostrar, fácilmente, que un amplio número de preguntas sobre programas son indecidibles
- Con nuestras definiciones, se puede enunciar como sigue:

Teorema de Rice

Si S es una propiedad semántica no trivial, entonces el problema de decidir si S se cumple para un programa P dado es indecidible.

Demostración del Teorema de Rice (I)

- Para demostrar el teorema de Rice, vamos a reducir el problema de la parada al problema de decidir si una propiedad semántica no trivial S se cumple para un programa P dado.
- Sea N un programa que no para con ninguna entrada ($\varphi_N(x) = \perp$)
- Hay dos posibilidades para S:
  - Caso 1: S se cumple para el programa N
  - Caso 2: S **no** se cumple para el programa N
- En el caso 1, dado que S es no trivial, debe haber otro programa para el cuál S **no** se cumple.
- En el caso 2, dado que S es no trivial, debe haber otro programa Q para el cuál S se cumple.

Demostración del Teorema de Rice (II)

- Tomaremos el caso 2 (la demostración para el caso 1 es análoga).
- Suponemos que el problema es resoluble, y por tanto podemos crear una macro $R(X)$ tal que $R(X) = 1$ si la propiedad semántica S se cumple para P_X, y $R(X) = 0$ si no.
- Ahora, dado el código c de un programa y una entrada k, construimos el siguiente programa:

```
begin
    X2:=U(c,k);
    X2:=0;
    Q
end
```

Demostración del Teorema de Rice (III)

- La función semántica de dicho programa d será:

$$\varphi_d(x) = \begin{cases} \varphi_Q(x) & \text{si } P_c \text{ para con entrada } k \\ \perp(\varphi_N(x)) & \text{en otro caso} \end{cases}$$

- Por el **Teorema de Parametrización**, sabemos que el código d del programa que calcula esta función semántica será una función total y computable $f(c, k)$ que depende de c y k.

Demostración del Teorema de Rice (IV)

- Por tanto, podemos crear este otro programa

```
begin
    X1:=f(X1,X2);
    X1:=R(X1)
end
```

- y su función semántica sería:

$$\varphi(c,k) = \begin{cases} 1 & \text{si la propiedad semántica } S \text{ se cumple para } P_d \\ 0 & \text{en otro caso} \end{cases}$$

Demostración del Teorema de Rice (V)

- Pero hemos visto que si P_c para con entrada k, P_d ejecuta Q, para el cuál sabemos que se cumple la propiedad semántica S
- Y en caso contrario, P_d ejecuta un programa equivalente a N, para el cuál hemos dicho que no se cumple.
- Por tanto, la función anterior se puede expresar como:

$$\varphi(c,k) = \begin{cases} 1 & \text{si } P_c \text{ para con entrada } k \\ 0 & \text{en otro caso} \end{cases}$$

- De nuevo, esta es la definición del **problema de la parada**
- Lo que demuestra que el problema es irresoluble.

Ejemplo con el Teorema de Rice (I)

- Recordemos el problema del ejemplo anterior

Nuestro problema

Dado un Programa While P, determinar si P devuelve 2 con al menos una entrada.

- Para aplicar el teorema de Rice deben cumplirse dos condiciones:
  1. Que el problema consista, de hecho, en determinar que dado un programa P, éste cumpla una cierta propiedad semántica S
  2. Que la propiedad semántica sea no trivial

Ejemplo con el Teorema de Rice (II)

- La propiedad que debemos determinar es "P devuelve 2 con al menos una entrada"
- Es, en efecto, una propiedad semántica, ya que solo depende de los valores devueltos por el programa.
- Para comprobar que es no trivial, basta con escribir un programa P_1 para el que la propiedad se cumple y otro programa P_2 para el que no.

Ejemplo con el Teorema de Rice (III)

- Un programa que **cumple** la propiedad sería, por ejemplo:

```
begin
    X1:=2
end
```

- Un programa que **no cumple** la propiedad sería, por ejemplo:

```
begin
    X1:=1
end
```

- Hemos demostrado que se trata de una propiedad semántica no trivial
- Por el teorema de Rice, el problema es **irresoluble**

Computabilidad y computadores reales

- En la Computabilidad trabajamos con modelos simples, versiones abstractas de computadores reales
- Sin embargo, sus propiedades esenciales son idénticas: máquinas que utilizan instrucciones (simples) sobre datos discretos, reciben entradas, hacen cálculos paso a paso y devuelven resultados
- Un computador real podría ser simulado, aunque muy lentamente, por una Máquina de Turing!!!

El problema de la parada y el teorema de Rice

- El estudio que hemos llevado a cabo sobre problemas irresolubles se puede adaptar fácilmente a computadores reales
- Tan solo hemos utilizado las ideas de codificación y de la existencia de una macro de universalidad
- Pero codificación y universalidad son elementos frecuentes en computadores reales (código binario, emuladores, máquinas virtuales, intérpretes...)
- Nuestra demostración de la irresolubilidad del problema de la parada puede aplicarse a la mayoría de los lenguajes modernos de programación con solo unos pocos cambios sintácticos
- Y, por tanto, empleando reducción también podemos demostrar algo similar al teorema de Rice
- Pero esta no es la única aplicación de estos métodos...

El antivirus perfecto

- Detectar *malware* y prevenir sus consecuencias es una tarea muy importante en la informática moderna
- ¿Pero, cuáles son las propiedades deseables de un antivirus *perfecto* ?
  - No debería decir que un programa que es un virus es inofensivo (no falsos negativos)
  - No debería decir que un programa que no es un virus es peligroso (no falsos positivos)
  - Debería parar siempre con cualquier entrada
  - **Debería no ser dañino en sí mismo**
- Desafortunadamente... un antivirus así es matemáticamente imposible!

El antivirus perfecto es imposible (I)

- Utilizaremos un enfoque parecido al que utilizamos para demostrar que el problema de la parada es indecidible (adaptado de la demostración original de Cohen, publicada en 1987)
- Entonces, suponemos que tenemos un procedimiento *AV* que implementa el antivirus perfecto (devuelve *Verdadero* si y solo si el programa a analizar es un virus)
- Consideramos el siguiente "programa":

```
if AV(ruta-al-programa) then
    imprimir("Tenga usted un buen día!")
else
    borrar todos los archivos e insultar al usuario
```

El antivirus perfecto es imposible(II)

- Ahora, considera qué pasaría si sustituyéramos la ruta a este programa *dentro de sí mismo* (creamos un nuevo programa que comprueba si él mismo es un virus)
- Si el procedimiento *AV* determinase que este nuevo programa es un virus entonces se comportaría inofensivamente (nótese que es importante que *AV* sea inofensivo). Esto es una contradicción.
- Pero si *AV* decide que el programa NO es un virus entonces hará algo muy maligno. De nuevo una contradicción!
- Por tanto, concluimos que el antivirus perfecto es imposible matemáticamente (tenemos que conformarnos con heurísticos!)

Problemas irresolubles en análisis sintáctico

- Quizás recuerdes de Teoría de Autómatas que algunos problemas sobre Gramáticas Libres de Contexto (GLCs) son indedicibles. Por ejemplo:
  - Determinar si una GLC es ambigua
  - Determinar si dos GLCs generan exactamente el mismo lenguaje
  - Determinar si una GLC genera todas las posibles cadenas sobre su alfabeto
  - Determinar si una GLC genera un lenguaje regular
  - ...
- Se puede demostrar que todos estos problemas son indecidibles mediante reducciones del conocido como Problema de Correspondencia de Post (véase el libro de Hopcroft, Ullman & Motwani)

- Quizás recuerdes (**deberías**) que determinar si una fórmula proposicional es una tautología es un problema decidible (se pueden utilizar tablas de verdad, p. ej.)
- Sin embargo, determinar si una fórmula de lógica de **predicados** es una tautología es ¡¡¡**indecidible**!!!
- Esto es, de nuevo, consecuencia de la indecibilidad del problema de la parada
- Dado el código p de un programa While y una entrada a, podemos escribir una fórmula de lógica de predicados $H_{p,a}$ que es válida si y solo si p para con entrada a.
- Hemos reducido el problema de la parada al problema de decidir la validez de fórmulas en lógica de predicados y, por tanto, ¡este último problema también es irresoluble!

Parte IV

Introducción a la complejidad algorítmica

Entre lo resoluble y lo irresoluble

- Un **problema de decisión** es **resoluble** o **decidible** si existe algún algoritmo que dé respuesta (afirmativa o negativa, según proceda) en un número **finito** de pasos.
- Si no existe, el problema se llama **irresoluble** o **indecidible**.
- Entre los decidibles y los indecidibles están los problemas **semi-decidibles** o **parcialmente decidibles**: aquellos para los que existe un algoritmo que da respuesta SI en un número finito de pasos para todas las entradas “positivas”, pero puede no parar para las de respuesta NO

Entre lo resoluble y lo irresoluble

- Ejemplos de problemas decidibles:
  - Dado un número natural n, ¿es n primo o no?
  - Dada una instancia P' de P ¿existe una solución S de P' tal que $Coste(S) \leq C$?
- Ejemplos de problemas semi-decidibles:
  - Dada una fórmula F en lógica de predicados, ¿es F satisfacible?
- Ejemplos de problemas indecidibles:
  - Dado un programa P, ¿parará para todas las entradas x?

Teoría de la Complejidad

- En la práctica, ¿vamos a poder dar soluciones a todos los problemas resolubles?

Teoría de la Complejidad

Estudia la dificultad de los problemas decidibles / resolubles (consumo de recursos)

- **Complejidad de un problema**: la del mejor algoritmo que lo resuelve (se conozca o no)
- Es muy difícil probar que un algoritmo es óptimo.
- En general, sólo tenemos cotas (que el mejor algoritmo conocido para P sea exponencial no implica que P lo sea)

Problemas tratables e intratables

- Sean P_1 y P_2 dos problemas decidibles: P_1 cuadrático en tiempo y P_2 exponencial.
- Supongamos que tenemos dos algoritmos A_1 y A_2 para resolverlos, de complejidades $t_1(n) = n^2$ y $t_2(n) = 2^n$ (por ejemplo)
- Comparemos tiempos de ejecución de ambos corriendo en una misma máquina capaz de ejecutar, por ejemplo, 10^6 operaciones/segundo.

Tamaño entrada	Tiempo aprox. A_1	Tiempo aprox. A_2
20	Milésimas de segundo	Segundos
100	Centésimas de segundo	Billones de siglos

Clases de complejidad

Clase de complejidad

Conjunto de problemas para los que existe un algoritmo que requiere, a lo sumo, un número "similar" de recursos en función del tamaño de la entrada.

- La definición formal requiere:
  - Modelo de computación
  - Modo (p.e. determinista/ no-determinista)
  - Recurso computacional a medir (p.e. tiempo/ espacio) .
- Las clases de complejidad agrupan los problemas en base a su dificultad (OJO: P_1 de $O(n) \sim P_2$ de $O(n^{1000})$)

Problemas *P* y *NP*

Problemas de la **clase P**:

- Se deciden/resuelven mediante un algoritmo determinista con complejidad acotada por un polinomio

Problemas de la **clase NP**:

- Existe un algoritmo **N**o determinista **P**olinomial que los resuelve (pueden resolverse en tiempo polinómico por una MT no determinista)
- Dado un "certificado", existe un algoritmo polinomial determinista que permite verificarlo
- Certificado: Información necesaria para verificar un resultado positivo
- *Ejemplo: En el SAT (decidir si una fórmula es satisfacible en L. de proposiciones), una interpretación I es un certificado*

Problemas P y NP

- Clases P y NP:
  - Está claro que $P \subseteq NP$
  - Problema abierto: ¿ $P = NP$?
- Ejemplos de problemas NP:
  - TSP (problema del viajante) en su versión de decisión (longitud k)
  - Problema del coloreado de grafos (K colores)
  - SAT: problema se SATisfabilidad de fórmulas en lógica de proposiciones
  - Problema de la suma nula de subconjuntos

Pertenencia a *NP*: Ejemplo

Problema de la suma nula de subconjuntos:

- "Dado un conjunto de números enteros, ¿Existe un subconjunto no vacío de ellos donde la suma de sus elementos es igual a 0?"
  - *p.e. ¿Existe un subconjunto del conjunto* {*-2, -3, 15, 14, 7, -10*} *tal que la suma de sus elementos sea 0? (instancia)*
- Encontrar el subconjunto puede llevar tiempo **¿P?**
- Comprobar que un subconjunto es solución es fácil (por ejemplo: {-2, -3, -10, 15}) y rápido (tiempo polinomial) $\Rightarrow$ **NP**

Problemas NP-completos

Problema NP-Completo

Un problema **Q** es **NP-completo** si:

$$Q \in NP \text{ y } \forall R \in NP, R \preceq Q$$

- Todo problema NP se puede transformar polinómicamente en **Q**:
- $R \preceq Q$ (**R** se reduce a **Q**) significa que existe un algoritmo polinomial determinista que a partir de un algoritmo para **Q** nos permite construir un algoritmo para **R**.
  - Si existiera un algoritmo polinomial para **Q** tendríamos un algoritmo polinomial para **R**

Problemas NP-completos

Q NP-completo: $Q \in NP$ y $\forall R \in NP, R \preceq Q$.

- **R** es más fácil de resolver que **Q**
- **Q** es más difícil de resolver que **R** (que todos los problemas NP)
- Nótese el paralelismo entre el método de reducción para demostrar la irresolubilidad y la forma de demostrar la complejidad **NP**.
- **El "puente" que permite la reducción aquí ha de ser un algoritmo polinomial**

Problemas NP-completos

- Todos los algoritmos conocidos para problemas **NP**-completos utilizan tiempo exponencial (eso no implica que sean exponenciales no polinomiales)
- Ejemplos de problemas **NP**-completos
  - TSP (problema del viajante) en su versión de decisión
  - Problema del coloreado de grafos (K colores)
  - SAT y variantes (3-SAT,...)
  - Problema de la mochila
  - Problema de la suma nula de subconjuntos
  - ...

Problemas NP-duros

Problema NP-Duro

Un problema **Q** es **NP-duro** si:

$$\forall R \in NP, R \preceq Q$$

- **NP**-completo = **NP** $\cap$ **NP**-duro
- Al menos tan difíciles como los **NP**-completos

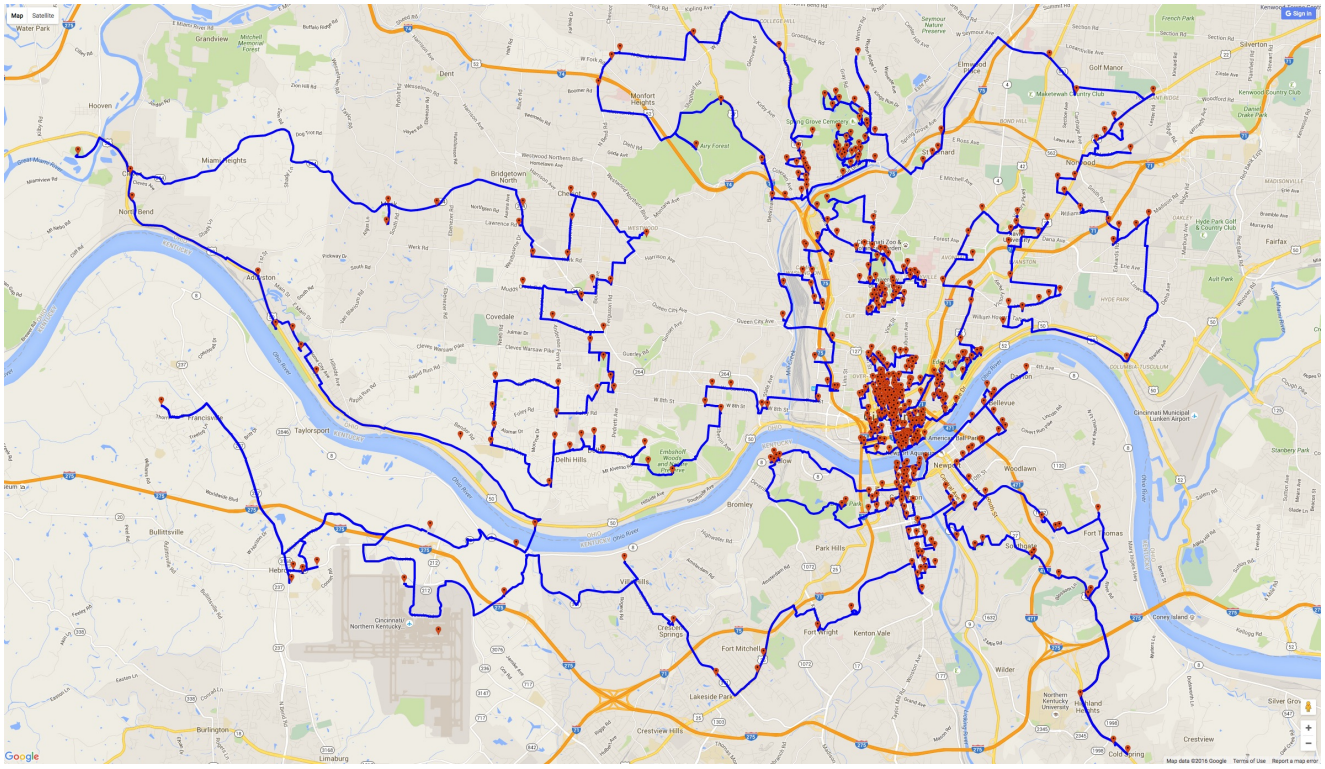
Referencia clásica: *Garey, M. and D. Johnson, Computers and Intractability; A Guide to the Theory of NP-Completeness, W. H. Freeman & Co.,1979*

El Problema del Viajante (TSP)

- Dados n lugares y la distancia entre cada par de ellos, encontrar un camino que visite todos los lugares una única vez y vuelva al punto de origen.
  - Problema de decisión: ¿Existe un un camino de longitud l, $l \leq K$?
  - Problema de optimización: ¿Cuál es el camino más corto?
- Verificar un certificado en el problema de decisión es fácil: Dado un recorrido, comprobar si visita todos los lugares sin repetir ninguno y tiene longitud $l \leq K$.
- Dado un certificado, comprobar si es **óptimo** no es fácil $\rightarrow$ el TSP no es NP
- No es NP-completo, pero es NP-duro

TSP: Aplicaciones

- Cuando se lanzó Pokémon Go, un periódico de Cincinnati publicó el mapa de las 551 Poképaradas y gimnasios de la ciudad
- y unos cuantos matemáticos aceptaron el desafío!



TSP: Aplicaciones

- Investigadores de la Universidad de Waterloo han calculado el recorrido mínimo para visitar todos los pubs de Reino Unido
- ...visitando 49.687 pubs y volviendo al inicio
- con un recorrido total de 63.739.687 m. (un sexto de la distancia a la luna)
- Veamos el recorrido!

El Teorema de Cook

Teorema de Cook

El problema SAT es **NP**-completo

- SAT es el problema de decidir si una fórmula de lógica de proposiciones en forma normal conjuntiva es satisfacible
- En la demostración se “simula”, mediante una fórmula de lógica proposicional en FNC, el funcionamiento de una Máquina de Turing no determinista (y que siempre para) sobre una entrada.
- La fórmula es satisfacible si y sólo si la MT acepta su entrada