Programación y Algorítmica Avanzadas

Curso 2025-26, cuatrimestre de primavera

José Luis Balcázar, Jordi Delgado

Departament de Ciències de la Computació, UPC

Resolución de algunos problemas

1. Consideramos la gramática con un único símbolo no terminal `S` y símbolos terminales `+`, `*`, `1` y `2`, y reglas `S` → `1`, `S` → `2`, `S` → `S + S`, `S` → `S * S`. Explica cómo derivar las palabras: `1 * 2`, `1 + 2 * 2` y `1 + 2 * 2 + 1`. Encuentra más ejemplos de palabras generadas.

Una posible solución al caso `1 + 2 * 2 + 1`:

`S` → `S + S` → `1 + S` → `1 + S * S` → `1 + 2 * S` → `1 + 2 * S + S`

→ `1 + 2 * 2 + S` → `1 + 2 * 2 + 1`

2. Algunas de esas palabras se pueden derivar de más de una manera. Encuentra derivaciones alternativas.

Una posible solución al caso `1 + 2 * 2 + 1`: el `S` inicial da lugar al `*` y los dos `S` junto al `*` dan lugar cada uno a un `+`:

`S` → `S * S` → `S + S * S` → `S + S * S + S` → ⋯ → `1 + 2 * 2 + 1`

3. ¿Cómo puedes argumentar que la palabra `1 + * 2` no se puede generar con esa gramática?

Argumentación: La única regla que permite insertar `+` coloca a su derecha `S`. La regla que permite insertar `*` coloca `S` a su izquierda. Tanto si primero se hace aparecer `+` y luego `*` como si se hace en el orden contrario, queda una `S` entre ambos. `S` no puede derivar en una palabra vacía porque ninguna regla lo permite: en todas la longitud de la palabra crece o queda igual. Por tanto, la `S` que queda entre ambos signos ha de derivar en algo y no es posible que queden adyacentes.

7. Construye, a partir de las funciones básicas y mediante composición y formación de pares, la función que hace corresponder a `x` el valor `<x.1>` (la podríamos llamar `piggyback_1`).

PReFScript: `pair id k_1`

10. Construye, a partir de las funciones básicas y mediante composición y formación de pares, la función *anterior* (modificado): `x - 1` para `x > 0` y `0` para `x = 0`.

PReFScript: `comp diff piggyback_1`

11. Explica por qué cualquier función constante está entre las funciones recursivas parciales.

Es pot demostrar per inducció:

La funció constant zero és `k_0 = compair diff k_1 k_1`, és a dir, `comp diff k_1_1` on `k_1_1` és `pair k_1 k_1`, per tant és una funció recursiva parcial.

(Farem servir quan convingui, a partir d'ara, `compair f g h` per a `comp f ff` on `ff` és `pair g h`. PReFScript ho permet si fem servir el `.pragma extended True`.)

Suposem que la funció constant k_{ct-1} on $ct > 0$ és recursiva parcial (hipòtesi d'inducció), anem a demostrar que k_{ct} ho és: k_{ct} és `compair add` k_{ct-1} `k_1`, per tant, en estar construida només amb les funcions bàsiques i una funció recursiva parcial, també és recursiva parcial.

12. Explica por qué está entre las funciones recursivas parciales, para cada natural `i`, la función que hace corresponder a `x` el valor `<x.i>`; idem para `<i.x>`.

PReFScript: Tenim (pel problema anterior) la funció constant `k_i` per a cada natural i. Puc fer les funcions `pair id k_i` i `pair k_i id`.

13. ¿Cómo se construyen las dos proyecciones π^L y π^R a partir de las funciones básicas?

PReFScript: Para la proyección izquierda, `pr_L(x): comp proj piggyback_0`, es decir, `pr_L(x): compair proj id k_0`. Para la derecha, `pr_R(x): comp s_tup piggyback_1`, es decir, `pr_R(x): compair s_tup id k_1`

14. ¿Y la función que a `x` le hace corresponder `<x.x>`?

PReFScript: `dupl: pair id id`

15. ¿Y la función que a x le hace corresponder x^2?

PReFScript: `comp mul dupl`

16. ¿Y la que a `<x.y>` le hace corresponder `<y.x>`?

PReFScript: Com que ja tenim `pr_L` i `pr_R`, `swap: pair pr_R pr_L`.

17. ¿Y la que a `<x.y>` le hace corresponder `<x+1.y>`?

PReFScript: `pair f pr_R` donde `f` es `compair add pr_L k_1`; `f` nos da `x+1` a partir de `<x.y>` y luego recuperamos la `y`.

18. Construye, a partir de las funciones básicas y mediante composición y formación de pares, la función signo: `sg(x) = 0 if x == 0 else 1` (será útil, entre muchos otros casos, para una de las posibles maneras de construir la operación de disyunción).

PReFScript: `sg: compair diff id ant`

19. Construye, a partir de las funciones básicas y mediante composición y formación de pares, las operaciones de negación, conjunción y disyunción.

(Revisa qué ocurre en todos estos casos cuando el valor `true` lo representan valores superiores a uno.)

negacio(x) $= 1-x$ és `compair diff k_1 id`; la conjunción puede ser `comp sign mul` (si está garantizado que `true` es 1 no es preciso el signo); para la disyunción podemos aplicar las leyes de de Morgan a las anteriores, o bien considerar `comp sign add`.

22. [2023-24] Construeix com a funció recursiva parcial la funció que, per l'argument `x`, retorna el par `<x.x-1>` (on cal recordar que la diferència és modificada i, per tant, `0 - 1` es considera `0` doncs no treballem amb negatius).

Es una parte de la solución a `sg` en el problema 18.

25. Argumenta que la funció següent és recursiva: la funció rep $x = \langle i.j \rangle$ i retorna com a imatge el quadrat de *la diferència usual dels enters* entre ells, $(i-j)^2$. *Ull viu*: el quadrat fa que el resultat sigui no negatiu encara que $i-j$ ho sigui; *no disposem* d'aquesta operació de diferència directament ja que $i-j$ sí pot ser negatiu.

Argumentació: Podem trobar el valor absolut de la diferència simplement com a la suma de les diferències modificades en ambdós sentits: si `d1` és `compair diff pr_L pr_R` i `d2` és `compair diff pr_R pr_L`, el valor absolut de la diferència és `compair add d1 d2` que finalment combinem amb la funció del problema 15.

27. Construye la función que, dados `x` e `y`, retorna el cociente de dividir `x` entre `y`; también para el resto; y, de nuevo, para obtener ambos a la vez, formando un par, pero usando *una única* minimización.

Para la división: si recibimos como dato `z = <x.y>` donde `x` es el dividendo e `y` el divisor, usamos minimización `mu` sobre una función que comprueba, dado el par `t = <z.q>` si `q` es válido como cociente. Para ello, comprobamos que `q + 1` es demasiado grande para ser el cociente; esto pasa para todos los valores superiores también, pero la minimización nos buscará el más pequeño, es decir, `q + 1` es demasiado grande pero `q` aún no. "Demasiado grande" significa que al multiplicar por `y` nos pasamos de `x`. Es decir, buscamos el `q` más pequeño tal que `y * (q + 1) > x`. Los valores de `y`, `q` y `x` se han de obtener de `t = <z.q> = <<x.y>.q>` por las oportunas proyecciones: `pr_R(pr_L(t))*(pr_R(t)+1) > pr_L(pr_L(t))`.

En PReFScript, una vez construida la función que evalúa esta expresión, basta ponerle un `mu` delante.

Para obtener cociente y resto con una sola minimización, es necesario buscar simultáneamente `<q.r>` comprobando que `y * q + r = x` y a la vez `r < x`.

35. Si f es (total y) biyectiva, su inversa está bien definida y es total. Si f es recursiva, ¿lo es f^{-1}?

Respuesta: sí; consideramos el predicado `test` que, dado `<x.y>`, devuelve 1 exactamente cuando $y = f(x)$. Este predicado se obtiene de f, que nos dicen que es recursiva, componiéndola con proyecciones y el test de igualdad. Entonces, `g: mu test` calcula la inversa de f buscando el menor (¡y único!) `y` que coincide con `f(x)`.

42. [2024-25] Sigui $f : \mathbb{N} \to \mathbb{N}$ una funció qualsevol (no necessàriament una funció recursiva parcial). Definim el conjunt $A_f = \{i \mid \phi_i = f\}$. Argumenteu que, o bé $A_f = \emptyset$ o bé A_f és infinit.

Tenim dos possibles casos:

f NO és recursiva parcial: Aleshores no existeix cap funció recursiva parcial ϕ_i tal que $\phi_i = f$. Això implica que $A_f = \emptyset$

f SÍ és recursiva parcial: Aleshores existeix una funció recursiva parcial ϕ_i tal que $\phi_i = f$. Però la funció recursiva parcial $id \circ \phi_i = \phi_i$, i té un índex diferent: Si l'índex de id és (tal com vam veure a les transparències) $\langle 0.0 \rangle = 1$, l'índex de $id \circ \phi_i$ és $\langle 1.\langle 1.i \rangle \rangle$, que és un nombre diferent d'i. El mateix podem argumentar per $id \circ (id \circ \phi_i) = id^2 \circ \phi_i = \phi_i$, etc.

En general, és clar que $id^n \circ \phi_i = \phi_i$ i $id^n \circ \phi_i$ té un índex diferent per a cada n. Per tant el nombre d'índexos de funcions que són iguals a ϕ_i és infinit.

43. [2024-25] Sigui el conjunt $A = \{x \mid \exists y \geq x \;\; \phi_y \text{ és constant}\}$. Què podem dir d'aquest conjunt? És decidible?

Hi ha infinits índexos per a cada funció recursiva parcial (veure problema anterior), per tant és clar que per a qualsevol x trobarem un $y \geq x$ tal que ϕ_y és constant. De fet, podem anar més enllà i afirmar que per a *qualsevol* $n \in \mathbb{N}$ podem trobar una funció recursiva parcial amb índex $y \geq x$ tal que $\forall m \; \phi_y(m) = n$, és a dir, $\phi_y =$ `k_n`. Én definitiva, $A = \mathbb{N}$ i per tant és trivialment decidible.

44. [2024-25] Sigui el conjunt $A = \{x \mid \phi_x = id\}$. És A decidible?

El primer que hem de fer és argumentar que el conjunt A és un conjunt *índex*.

Si tenim dues funcions ϕ_i i ϕ_j tal que $\phi_i = \phi_j$, és obvi que, o bé $i, j \in A$ (és a dir, $\phi_i = \phi_j = id$) o bé $i, j \notin A$, ja que en ser iguals les funcions, la propietat de ser igual a id de les funcions que caracteritza A o bé es verifica en les dues funcions ϕ_i i ϕ_j o bé no es verifica en cap.

Ara cal veure que A no és *trivial*, és a dir, $A \neq \emptyset$ i $A \neq \mathbb{N}$. En aquest cas és fàcil,

- És obvi que la funció recursiva parcial $id \in A$ (recordem que l'índex de id és 1) per tant, $1 \in A$. Així doncs $A \neq \emptyset$.
- La funció ϕ_0, que vam identificar amb la funció indefinida a tot arreu ($\phi_0(x) \uparrow$ per a tot x), és diferent de la funció id, per tant $0 \notin A$ i $A \neq \mathbb{N}$.

Ara ja podem aplicar el teorema de Rice per afirmar que A no és decidible.

45. [2022-23] Considerem el conjunt $A = \{i \,|\, \phi_i(42) \downarrow\}$, és a dir, els índexos i tals que el valor 42 és al domini de la funció ϕ_i. Argumenta que és indecidible.

És clar que A és un conjunt índex: si $\phi_i = \phi_i$, aleshores el domini es el mateix i, si 42 hi és, llavors la funció hi està definida i tots dos valors i i j pertànyen a A; en cas contrari, cap dels dos hi pertany. Apliquem el teorema de Rice: 42 no és al domini de la funció completament indefinida, però, sí és al domini de qualsevol funció total (la identitat, les constants...); per tant, A no és buit ni conté tots els naturals i el teorema de Rice ens diu que aleshores és indecidible.

46. [2024-25] Sigui el conjunt $A = \{i | \phi_i(y) = y$, per a infinits $y\}$. Argumenta que A és indecidible.

Aplicarem el teorema de Rice. És clar que A és un conjunt índex: Si $\phi_i = \phi_i$, aleshores o bé tots dos pertànyen a A o bé cap dels dos ja que les dues funcions iguals verifiquen totes dues la condició per a que el seu índex pertànyi a A, o no la compleixen les dues. Ara, veiem que A és no trivial: l'índex d'`id` (funció recursiva parcial bàsica identitat) pertany a A, per tant A no és buit. L'índex de `compair add id k_1` (és a dir, $f(x) = x + 1$, recursiva total per construcció) no pertany a A, per tant A no és igual a tots els naturals.

48. [2024-25] Argumenteu (o demostreu!) que $A \leq_m^p B \Leftrightarrow \overline{A} \leq_m^p \overline{B}$

Suposem que $A \leq_m^p B$, això vol dir que existeix una funció recursiva total f, calculable en temps polinòmic, tal que $x \in A \Leftrightarrow f(x) \in B$. Per tant:

$$x \in \overline{A} \Rightarrow x \notin A \Rightarrow f(x) \notin B \Rightarrow f(x) \in \overline{B}$$

$$x \notin \overline{A} \Rightarrow x \in A \Rightarrow f(x) \in B \Rightarrow f(x) \notin \overline{B}$$

Per tant $\overline{A} \leq_m^p \overline{B}$

Argumentar que $\overline{A} \leq_m^p \overline{B} \Rightarrow A \leq_m^p B$ es fa igual, el raonament és el mateix.

49. [2024-25] Demostra la transitivitat de $\leq_m^p$. És a dir, si $A \leq_m^p B$ i $B \leq_m^p C$, aleshores $A \leq_m^p C$.

$A \leq_m^p B$, vol dir que existeix una funció recursiva total f, calculable en temps polinòmic, tal que $x \in A \Leftrightarrow f(x) \in B$, i $B \leq_m^p C$, vol dir que existeix una funció recursiva total g, calculable en temps polinòmic, tal que $x \in B \Leftrightarrow g(x) \in C$.

És clar que la funció $g \circ f$ verifica:

$$x \in A \Leftrightarrow f(x) \in B \Leftrightarrow g(f(x)) \in C \equiv (g \circ f)(x) \in C$$

Ara cal argumentar que:

- $(g \circ f)$ és recursiva total. Per construcció és així, ja que f i g ho són.
- $(g \circ f)$ és calculable en temps polinòmic? Ara ho veiem...

Sabem que donat un x, $f(x)$ és calcula en temps polinòmic en $|x|$: $p(|x|)$ on p és un polinomi. $(g \circ f)(x) = g(f(x))$ és calcula en temps polinòmic en $|f(x)|$: $t(|f(x)|)$ on t és un polinomi (ja que g és calculable en temps polinòmic).

Quina mida té $f(x)$? És clar que no pot ser més gran que un polinomi en $|x|$, ja que en un nombre polinòmic de passos (temps $p(|x|)$) no pot escriure una sortida més gran que un nombre polinòmic de caselles: $|f(x)| \sim q(|x|)$ on q és un polinomi.

Així doncs, $t(|f(x)|) \sim t(q(|x|))$ i, com la composició de polinomis és un polinomi, és clar que $(g \circ f)$ és una funció recursiva total, calculable en temps polinòmic $(t \circ q)$ (en $|x|$).

52. [2022-23] Considerem els següents conjunts:
$$A = \{(M, p_0, \ldots, p_{n-1}) \mid \exists J \subseteq \{0, \ldots, n-1\} \sum_{i \in J} p_i = M\}$$
$$B = \{(x_0, \ldots, x_{m-1}) \mid \exists J \subseteq \{0, \ldots, m-1\} \sum_{i \in J} x_i = \sum_{i \notin J} x_i\}$$
on M, els p_i i els x_i són tots enters no negatius. Podem veure que el primer és la variant de la Motxilla que vam anomenar a classe *Subset Sum*, només amb pesos i tot requerint emplenar exactament la motxilla. El segon es coneix com a *Partició*.
Argumenta que $A \leq_m^p B$: per fer això, indica com construir valors x_i per B, a partir de valors M i p_i per A, de manera que $(M, p_0, \ldots, p_{n-1})$ és a A si i només si $(x_0, \ldots, x_{m-1})$ és a B, sense oblidar argumentar que es poden construir en temps polinòmic. Qué en pots deduir immediatament sobre B? Argumenta que la reducció en el sentit contrari també existeix, però, *sense construir-la explícitament*. En dedueixes quelcom?

Donat M i els p_i, trobem $S = \sum p_i$ i fem $x_i = p_i$, $x_n = M+1$ i $x_{n+1} = S+1-M$. Es poden calcular en temps polinòmic. Si el cas de *Subset Sum* té solució J, $J \cup \{n+1\}$ suma $S+1$ i la resta, amb x_n, també. Si el cas de Partició té solució J, primer de tot veiem que J no conté simultàniament n i $n+1$: x_n i x_{n+1} sumen ja $S+2$ i amb tota la resta no hi ha prou per igualar-lo. (Sense els $+1$ a tots dos valors no es pot descartar aquesta opció, de fet, sense els $+1$ la partició existeix sempre.) Ara, com que la suma de tots els x és $2S+2$, la suma dels elements indexats per J i també la de la resta són $S+1$. Al costat on és $n+1$, la resta ha de sumar M.

Ull viu: si només hi afegim un $x_n = S - 2M$ sembla que funciona; però, aquest valor pot ser negatiu, i l'enunciat no ho permet.

Se'n pot deduir que és NP-hard. Res més.

La reduccio en sentit contrari es demostra tot argumentant que Partició és a NP i aplicant la NP-completesa de Subset Sum: triem indeterminísticament els indexos J i comprovem que la suma dels elements a J i la suma de la resta

coincideixen. Ara, com que sabem que *Subset sum* és NP-complet, ja sabem, sense construir la reducció, que *Partition* es redueix a *Subset sum*. Ara sí és quan podem deduir que és NP-complet.

54. [2024-25] Sabent que `pair` = $\lambda x.\lambda y.\lambda p.pxy$, i `first` = $\lambda p.p(\lambda x.\lambda y.x)$, calcula (la forma normal de) `first pair 2 1`. No cal saber la codificació en λ-càlcul del 2 ni de l'1 per fer aquest problema.

Sabem que `pair 2 1` és $\lambda x.\lambda y.\lambda p.pxy$ `2 1` que redueix a $\lambda p.p$ `2 1`. Aleshores, `first pair 2 1` és $\lambda p.p(\lambda x.\lambda y.x)(\lambda p.p$ 2 1) o $\lambda p.p(\lambda x.\lambda y.x)(\lambda t.t$ 2 1) per α-reducció; d'ací $(\lambda t.t$ 2 1$)(\lambda x.\lambda y.x)$, o sigui $(\lambda x.\lambda y.x)$ 2 1 que és 2, com esperavem.

55. [2024-25] Recordem que, en el λ-càlcul, `TRUE` es representa per λxy.x i `FALSE` per λxy.y. Argumenta per què podem definir `OR` p q com p p q (o el que és el mateix, `OR` = λp.λq.p p q)

La clau està en veure que `TRUE` i `FALSE` són funcions de dos paràmetres, dels que `TRUE` retorna el primer i `FALSE` retorna el segon.

Fixem-nos que:

- `OR TRUE` y = `TRUE` **`TRUE`** y = `TRUE`.
- `OR FALSE TRUE` = `FALSE FALSE` **`TRUE`** = `TRUE`.
- `OR FALSE FALSE` = `FALSE FALSE` **`FALSE`** = `FALSE`.

56. [2022-23] Sabent que en el λ-càlcul `TRUE` es representa per λxy.x i `FALSE` per λxy.y, argumenta en detall que λab.ab(λxy.y) és la implementació de la funció `AND`.

Una opció és completar la taula de veritat, tot aplicant el λ-terme als quatre parells i operant amb β-reduccions. Un dels quatre casos, `False and False = False`, és:

λ`uv.uv(`λ`xy.y)` λ`xy.y` λ`xy.y` = λ`xy.y` λ`xy.y` λ`xy.y` = λ`xy.y`

Els altres tres son semblants. Una altra opció és veure que si fixem el primer argument a `True` ens queda la funció identitat, i que si el fixem a `False` ens queda la funció constant `False`:

λ`uv.uv(`λ`xy.y)` λ`xy.x` = λ`v.` λ`xy.x v` λ`xy.y` = λ`v.v`

λ`uv.uv(`λ`xy.y)` λ`xy.y` = λ`v.` λ`xy.y v` λ`xy.y` = λ`v.` λ`xy.y`

58. [2024-25] A classe vam veure el combinador de punt fix `Y` descobert pel lògic Haskell B. Curry, i vam veure que per a qualsevol λ-expressió `R` es verifica que `Y R = R (Y R)`. Alan Turing també va descobrir un combinador de punt fix $\Theta = (\lambda$xy.y (x x y))(λxy.y (x x y)). Demostreu que per a qualsevol λ-expressió `R` es verifica que Θ `R = R (`Θ `R)`

Fixem-nos que, si definim `A` $\equiv (\lambda$xy.y (x x y)), aleshores Θ = `A A`. Per tant: Θ `R` = `A A R`.

A partir d'aquí, centrant-nos només en una $\mathtt{A}$:

$\Theta\ \mathtt{R} = \mathtt{A}\ \ \mathtt{A}\ \mathtt{R} = (\lambda \mathrm{xy.y\ (x\ x\ y)})\ \mathtt{A}\ \mathtt{R} \rightarrow_\beta (\lambda \mathrm{y.y\ (}\mathtt{A}\ \ \mathtt{A}\ \mathrm{y)})\ \mathtt{R} \rightarrow_\beta \mathtt{R}\ (\mathtt{A}\ \ \mathtt{A}\ \mathtt{R}) = \mathtt{R}\ (\Theta\ \mathtt{R})$

i ja està.