Programación para la IA

Bibliografía:

- Apuntes de clase.
- Wikipedia, the Free Encyclopedia. *Reduction Strategy*. Recuperado de https://en.wikipedia.org/wiki/Reduction_strategy el 25 de septiembre de 2022.
- Simon Marlow (ed.) *Haskell 2010 Language Report* (2010). Recuperado de https://www.haskell.org/onlinereport/haskell2010/haskell.html.
- Haskell Wiki. *Monad laws*. Recuperado de https://wiki.haskell.org/Monad_laws el 5 de diciembre de 2022.

Capítulo 1

Introducción

1.1. Historia

En 1928, David Hilbert planteó el **problema de la decisión** o ***Entscheidungsproblem***, que pide un algoritmo para determinar si una proposición lógica se deriva o no de los axiomas en un sistema lógico que incluya la aritmética de Peano. En 1931, Gödel demuestra en su **teorema de incompletitud** que tal sistema lógico no puede ser a la vez consistente y completo. En 1936, Alan Turing y Alonzo Church, de forma independiente, crean respectivamente las **máquinas de Turing** y el **cálculo** λ para definir el concepto de algoritmo o **función computable** y lo usan para probar que no hay algoritmo que resuelva el *Entscheidungsproblem*.

Las máquinas de Turing fundamentan la teoría de la computabilidad, y el cálculo λ la de la programación funcional. Moses Shönfinkel y Haskell Curry introducen la **lógica combinatoria**, útil en la implementación de lenguajes funcionales.

En 1956, John McCarthy organiza el *Dartmouth Summer Research Project on Artificial Intelligence*, que populariza el término «inteligencia artificial», y en los años sucesivos Newell, Simon y Shaw escribieron «*Logic theorist*» y «*General problem solver*» y McCarthy escribió «*Programs with common sense*». En 1958, McCarthy crea el lenguaje **Lisp** (*LISt Processing*) para usarlo en inteligencia artificial. Este usa listas[1], funciones de orden superior, recursividad y recolección de basura, aunque también características de lenguajes imperativos como asignación destructiva y efectos colaterales para hacerlo práctico, y fue el primer lenguaje de alto nivel interpretado. Posteriormente surgió Scheme, basado en Lisp y más cercano a la idea original de Lisp. Lisp aparece en la mayoría de la literatura de IA y se ha usado en Carnegie Mellon, MIT y Stanford. Los lenguajes **ISWIM** (*If you See What I Mean*) se acercan más al cálculo λ que Lisp.

En 1964, Peter Landin crea una **semántica denotacional** para un subconjunto del lenguaje imperativo ALGOL 60, relacionándolo con el cálculo λ.

A principios de los 70 se da la crisis del software y se busca facilitar la verificación formal de programas con modelos como la **programación funcional** o **aplicativa** y la **programación lógica** o **declarativa**.

[1] En las diapositivas dice que como tipo básico, pero en realidad están hechas por pares («conses»), que sí son un tipo básico.

En 1978, John Backus publica «*Can programming be liberated from the von Neumann style?*», donde critica la programación imperativa, muestra las ventajas de la programación funcional y diseña el lenguaje FP (*Functional Programming*), con la idea de definir nuevas funciones combinando otras. Esta adquiere interés en varias universidades británicas, y se crean lenguajes funcionales como Hope, ML, SASL, KRC o Miranda, que usan conceptos como polimorfismo, evaluación perezosa y funciones de orden superior y consiguen código muy abstracto y muy conciso.

En septiembre de 1987 se celebra la conferencia **FPCA**, en la que se discuten los problemas de esta proliferación de lenguajes funcionales y se forma un comité internacional para diseñar un lenguaje puramente funcional de propósito general, **Haskell**, que busca unificar las características más importantes de los lenguajes funcionales. Durante casi 10 años aparecieron versiones del lenguaje hasta que en 1998 se crea el estándar **Haskell 98**, que se continúa con la investigación de nuevas características y extensiones, llegando a la última versión del estándar, **Haskell 2010**. Actualmente proliferan lenguajes multiparadigma que tienen base imperativa pero incorporan conceptos de programación funcional, como **Python**, creado a finales de los 80, y lenguajes funcionales como Haskell se usan para muchas aplicaciones.

1.2. Programación funcional

Un **programa imperativo** es una secuencia de **comandos** que modifican un **estado**. El principal modificador es la asignación, $v = E$ o $v := E$. Se pueden ejecutar comandos en secuencia escribiéndolos separados, por ejemplo, por «;», y ejecutarse condicionalmente con `if` o `while`. Los **lenguajes imperativos**, como FORTRAN, C, ALGOL o Java, soportan programación imperativa.

Este modelo establece una dependencia entre el algoritmo y el modelo von Neumann, donde los datos del programa corresponden a datos en memoria principal y el código a instrucciones máquina en dicha memoria, lo que permite generar programas eficientes y sencillos al tener un nivel cercano a la máquina pero dificulta usar los algoritmos en arquitecturas que usen otros modelos.

Además, las modificaciones del estado con asignaciones oscurecen la semántica del programa, dificultan estudiar la corrección y complican la optimización y paralelización del código por parte del compilador.

Un **programa funcional** es una expresión, y ejecutar el programa significa evaluar la expresión. No hay estado ni asignaciones, ni secuenciación o repetición ya que la evaluación de una expresión no afecta a otras, pero los usos de estas se pueden simular mediante recursividad. Se usan funciones en sentido matemático, cuyo valor devuelto depende sólo de los parámetros de entrada, y se tiene **transparencia referencial**, consistente en que una misma expresión siempre evalúa al mismo valor, lo que facilita probar la corrección de programas. Las funciones se pueden usar como valores, permitiendo **funciones de orden superior**, que aceptan otras funciones como parámetros o devuelven una función, dando flexibilidad al programador.[2]

Los **lenguajes funcionales** soportan programación funcional, y suelen incluir:

- **Inferencia de tipos**, con la que el programador no tiene que declarar el tipo de los valores en la mayoría de los casos sino que el compilador lo deduce, aunque el programador puede declarar el tipo para que el compilador lo compare con el tipo inferido.

[2]Las diapositivas llaman función de orden superior a cualquiera que se use como valor. Esto es incorrecto.

Esto aumenta la concisión respecto a los lenguajes en los que hay que declarar los tipos explícitamente, y da mayor seguridad y eficiencia respecto a los lenguajes con tipos dinámicos ya que evita tener que comprobar los tipos en tiempo de ejecución y los errores que aparecen cuando estas comprobaciones fallan, ya que las comprobaciones se han hecho al compilar.

- **Polimorfismo**, con el que el tipo de los parámetros o la salida de una función dependen de uno o más parámetros, permitiendo una mayor reutilización del código al no tener que repetir algoritmos para estructuras similares.

- **Evaluación perezosa**, no evaluando un argumento hasta que se necesita, evaluando después de sustituirlo en la definición de la función, aunque guardando el valor evaluado por eficiencia. Esto es en contraposición a la tradicional **evaluación ansiosa**, en que primero se evalúan los argumentos y luego se llama a la función. La evaluación perezosa permite manipular estructuras de datos conceptualmente infinitas.

Como no tienen variables ni asignación, los programas son más cercanos a objetos matemáticos abstractos y aumenta la libertad de implementación, permitiendo la paralelización. Además, las funciones de orden superior hacen el código más conciso y elegante, mejoran la reusabilidad y modularidad y permiten representar estructuras de datos infinitas de forma conveniente. También hay herramientas para probar la corrección de programas funcionales, lo que en lenguajes imperativos es complicado porque hay que explicitar los estados.

Por otro lado, es más difícil representar entrada y salida y programas interactivos o que se ejecutan de forma continua, como editores o controladores, y al estar más alejados del *hardware* que los lenguajes imperativos, son menos eficientes y es más difícil razonar sobre su eficiencia en tiempo y espacio.

La **notación lambda** representa una función $x \mapsto E[x]$ como $\lambda x.E[x]$, que también se puede escribir como $[x]\ E[x]$. Una función $f : X_1 \times \cdots \times X_n \to Y$ se puede representar como $g : X_1 \to \cdots \to X_n \to Y$, donde «$\to$» es asociativo por la derecha, de modo que $g(x_1)(x_2)\cdots(x_n) = f(x_1, \ldots, x_n)$. A esto se le llama **currificación** en honor a Haskell Curry.

Capítulo 2

Cálculo lambda

Un **sistema formal** está formado por:

1. Un **lenguaje formal**, con una serie de **símbolos** y **reglas sintácticas** para combinarlos obteniendo **fórmulas**.
2. Una serie de **reglas de inferencia**, formadas por 0 o más juicios en el antecedente (arriba) y uno en el precedente (abajo). Los juicios son expresiones de la forma $\Gamma \vdash f$, que indican que la fórmula f se deriva de las fórmulas en Γ, donde f puede contener símbolos que indican entidades de cierto tipo en el lenguaje (subcadenas con ciertas restricciones) y el mismo símbolo representa la misma entidad en toda la regla.

Un **teorema** es una fórmula f tal que $\vdash f$ se puede obtener aplicando una cantidad finita de reglas de inferencia. Un **axioma** es un teorema que se obtiene aplicando una sola regla de inferencia.

El **cálculo lambda** es un sistema formal diseñado por Alonzo Church y Stephen Kleen en los años 30[1] para investigar la definición de funciones computables, aplicación de funciones y recursividad. Es el precursor de lenguajes funcionales como Lisp, ML y Haskell, y puede expresar cualquier función computable.

El alfabeto del lenguaje está formado por la **igualdad** $=$, la **reducción** $\to$, los **símbolos impropios** λ, ., (y), y símbolos para representar una cantidad infinita de variables. Un **término** es una variable, una **aplicación** $(M\,N)$, donde M y N son términos, o una **abstracción** $(\lambda x.M)$, donde x es una variable llamada **variable ligada** y M es un término llamado **cuerpo**. Una fórmula es una expresión $M = N$ o $M \to N$, donde M y N son términos. Dos términos M y N son **idénticos**, $M \equiv N$, si tienen la misma expresión.

Para reducir el uso de paréntesis, consideramos que la aplicación tiene máxima prioridad y es asociativa por la izquierda, la abstracción se extiende lo más lejos posible y $\to$ y $=$ tienen el máximo alcance. Además, si $x_1, \ldots, x_n$ son variables y M es un término, la sintaxis $\lambda x_1\, x_2\, \ldots\, x_n.M$ equivale a $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.M$.

Si x es una variable y M y N son términos, definimos el conjunto de **variables libres** de un término como

$$\mathrm{FV}(x) \coloneqq \{x\}, \qquad \mathrm{FV}(M\,N) \coloneqq \mathrm{FV}(M) \cup \mathrm{FV}(N), \qquad \mathrm{FV}(\lambda x.M) \coloneqq \mathrm{FV}(M) \setminus \{x\},$$

[1] Los del siglo XX.

y el conjunto de **variables ligadas** como

$$\mathrm{BV}(x) \coloneqq \emptyset, \qquad \mathrm{BV}(M\,N) \coloneqq \mathrm{BV}(M) \cup \mathrm{BV}(N), \qquad \mathrm{BV}(\lambda x.M) \coloneqq \mathrm{BV}(M) \cup \{x\}.$$

Estos conjuntos no contienen variables sino apariciones de variables en la expresión, y en sus definiciones para una abstracción, el término $\{x\}$ se refiere a todas las apariciones de x en M.

Sean M, N y P términos y x e y variables distintas, definimos la **sustitución** de un término en otro como

$$\begin{aligned} x[N/x] &\coloneqq N, \\ y[N/x] &\coloneqq y, \\ (M\,P)[N/x] &\coloneqq (M[N/x])(P[N/x]), \\ (\lambda x.M)[N/x] &\coloneqq \lambda x.M, \\ (\lambda y.M)[N/x] &\coloneqq \lambda z.(M[z/y][N/x]), \end{aligned}$$

donde z es una variable cualquiera que no está en $\mathrm{FV}(M) \cup \mathrm{FV}(N)$. Si $y \notin \mathrm{FV}(N)$ o $x \notin \mathrm{FV}(M)$, en la última regla se puede omitir $[z/y]$ y la sustitución es **válida** o **segura**, pero si no es así, al aplicar la versión simplificada se dice que se produce **captura de variables**, en la que una variable libre en N pasa a estar ligada en $(\lambda y.M)[N/x]$.

Las reglas de inferencia son las siguientes, donde M, N, S, T y U son términos y x e y son variables.

$$\alpha\text{-conversión: } \frac{\{y \notin \mathrm{FV}(M)\}}{\Gamma \vdash (\lambda x.M) \to (\lambda y.M[y/x])} \qquad \frac{\Gamma \vdash S \to T}{\Gamma \vdash (U\,S) \to (U\,T)}$$

$$\beta\text{-conversión: } \frac{}{\Gamma \vdash ((\lambda x.M)\,N) \to M[N/x]} \qquad \frac{\Gamma \vdash S \to T}{\Gamma \vdash \lambda x.S \to \lambda x.T}$$

$$\eta\text{-conversión: } \frac{\{x \notin \mathrm{FV}(M)\}}{\Gamma \vdash (\lambda x.(M\,x)) \to M} \qquad \frac{\Gamma \vdash S \to T}{\Gamma \vdash S = T}$$

$$\frac{}{\Gamma \vdash T \to T} \qquad \frac{\Gamma \vdash S = T}{\Gamma \vdash T = S}$$

$$\frac{\Gamma \vdash S \to T \quad \Gamma \vdash T \to U}{S \to U} \qquad \frac{\Gamma \vdash S = T \quad \Gamma \vdash T = U}{\Gamma \vdash S = U}$$

$$\frac{\Gamma \vdash S \to T}{\Gamma \vdash (S\,U) \to (T\,U)}$$

Las **conversiones** α, β y η también se llaman **reducciones**. El resto de reglas dicen que las reducciones se pueden aplicar a subtérminos, que la reducibilidad es reflexiva y transitiva, y que dos términos son iguales si se pueden conectar por una cadena finita de reducciones hacia un lado u otro. La igualdad es **extensional**, de modo que si dos funciones son iguales, al aplicarlas a los mismos parámetros se obtienen resultados iguales.

El **cálculo lambda puro** es el que se ha descrito, pudiendo definir términos comunes para abreviar pero sin constantes predefinidas. El **cálculo lambda impuro** permite también constantes predefinidas sin definir estas como términos, como números o funciones sobre estos. Se añaden **δ-reducciones**, en las que una expresión en que intervienen constantes se sustituye por su valor según reglas específicas a dichas constantes.

Un **radical** o ***redex*** es un término de la forma $(\lambda x.U)T$, y su **contrato** es $U[T/x]$. Un término M es **normal** si no contiene ningún radical, y es **normalizable** si existe un término

normal N tal que $M \to N$, en cuyo caso se puede llegar a una por una cadena de β-reducciones en el término o subtérminos. El objetivo principal al manipular una expresión lambda es reducirla a su forma normal, lo que corresponde a la idea de fin de cálculo en la programación convencional. Hay expresiones no normalizables, como $(\lambda x.xx)\lambda x.xx$.

Una **estrategia de evaluación** indica el orden en que aplicar β-reducciones para tratar de llegar a la forma normal. Algunas son:

1. **Orden normal:** Se reduce primero el radical más exterior más a la izquierda, que para un término $(\lambda x.S)T$ es el propio término, para otra aplicación $S\,T$ es el de S si tiene alguno o el de T en otro caso, y para $\lambda x.S$ es el de S.[2]

2. **Orden aplicativo:** Se reduce primero el radical más interior más a la izquierda, que se define como el radical exterior más a la izquierda salvo que, para un término $(\lambda x.S)T$, primero se reduce el primero de S si tiene alguno, o el primero de T si tiene alguno y S no.[3]

Teoremas de Church-Rosser:

1. **Propiedad del diamante:** Si $S \to T_1$ y $S \to T_2$, existe U tal que $T_1 \to U$ y $T_2 \to U$. Además, la forma normal de un término a la que se llega por β-reducciones es única salvo α-conversiones.

2. Si un término es normalizable, se puede llevar a su forma normal por el orden de reducción normal.

Un **combinador** o **término cerrado** es un término sin variables libres. Los más simples son $\mathbf{I} \coloneqq \lambda x.x$, $\mathbf{K} \coloneqq \lambda x\,y.x$ y $\mathbf{S} \coloneqq \lambda f\,g\,x.f\,x\,(g\,x)$. Todo término tiene un equivalente con las mismas variables libres y sin abstracciones usando solo estos 3 combinadores.

[2] Es lo que usan los lenguajes perezosos como Haskell, aunque con optimizaciones.

[3] Es lo que usan la mayoría de lenguajes de programación, salvo que no se reduce el interior de una expresión lambda y en algunos no se garantiza que la evaluación de parámetros se haga de izquierda a derecha.

Capítulo 3

Programación en cálculo lambda

Definimos

$$\begin{aligned}
\texttt{true} &\coloneqq \lambda x\, y.x,\\
\texttt{false} &\coloneqq \lambda x\, y.y,\\
\texttt{cond} &\coloneqq \lambda i\, t\, e.i\, t\, e = \lambda x.x,\\
\texttt{and} &\coloneqq \lambda x\, y.x\, y\, \texttt{false},\\
\texttt{or} &\coloneqq \lambda x\, y.x\, \texttt{true}\, y = \lambda x.x\, \texttt{true},
\end{aligned}$$

y escribimos $\texttt{if}\, i\, \texttt{then}\, t\, \texttt{else}\, e \coloneqq \texttt{cond}\, i\, t\, e$, $a \wedge b \coloneqq \texttt{and}\, a\, b$, $a \vee b \coloneqq \texttt{or}\, a\, b$ y $\neg a \coloneqq a\, \texttt{false}\, \texttt{true}$.

Dado un tipo de dato $(X_{11} \times \cdots \times X_{1k_1}) \sqcup \cdots \sqcup (X_{m1} \times \cdots \times X_{mk_m})$, podemos representar $(x_1, \ldots, x_{k_i}) \in X_{i1} \times \cdots \times X_{ik_i}$ como $\lambda f_1 \cdots f_m.f_i\, x_1 \ldots x_{k_i}$. Así se definen los pares ordenados:

$$(E_1, E_2) \coloneqq \lambda f.f\, E_1\, E_2.$$

Podemos acceder a los miembros con los **destructores**

$$\begin{aligned}
\texttt{fst} &\coloneqq \lambda p.p\, \texttt{true},\\
\texttt{snd} &\coloneqq \lambda p.p\, \texttt{false}.
\end{aligned}$$

Representamos una tupla de $n \geq 2$ elementos como

$$(E_1, \ldots, E_n) \coloneqq (E_1, (E_2, \cdots (E_{n-1}, E_n) \cdots)),$$

y las **proyecciones**

$$\begin{aligned}
(p)_1 &\coloneqq \texttt{fst}\, p,\\
(p)_n &\coloneqq \texttt{snd}^{n-1}\, p,\\
(p)_i &\coloneqq \texttt{fst}(\texttt{snd}^{i-1}\, p), \qquad\qquad 2 \leq i < n,
\end{aligned}$$

donde $f^0 \coloneqq \lambda x.x$ y, para $n > 0$, $f^n \coloneqq \lambda x.f(f^{n-1}\, x)$.

Para $n \geq 2$,

$$\begin{aligned}
\texttt{CURRY}_n f &\coloneqq \lambda x_1 \cdots x_n.f(x_1, \ldots, x_n),\\
\texttt{UNCURRY}_n f &\coloneqq \lambda p.g\, (p)_1 \cdots (p)_n,
\end{aligned}$$

y escribimos $\lambda(x_1, \ldots, x_n).T \coloneqq \texttt{UNCURRY}_n(\lambda x_1 \cdots x_n.T)$. La **$\beta$-conversión generalizada** afirma que $(\lambda(x_1, \ldots, x_n).T[x_1, \ldots, x_n])(t_1, \ldots, t_n) = T[t_1, \ldots, t_n]$.[1]

Representamos $n \in \mathbb{N}$ con el **número** o **entero de Church** $n \coloneqq \lambda f.f^n$.

$$\begin{aligned}
\texttt{succ} &\coloneqq \lambda n\, f\, x.n\, f(f\, x) = \lambda n\, f\, x.f(n\, f\, x),\\
\texttt{iszero} &\coloneqq \lambda n.n(\lambda x.\texttt{false})\texttt{true},\\
(+) &\coloneqq \lambda m\, n\, f\, x.m\, f(n\, f\, x),\\
(\cdot) &\coloneqq \lambda m\, n\, f.m(n\, f),\\
(\hat{}) &\coloneqq \lambda m\, n.n\, m,
\end{aligned}$$

y escribimos $a + b \coloneqq (+)a\, b$, $a \cdot b \coloneqq (\cdot)a\, b$ y $a^b \coloneqq (\hat{})a\, b$.

Igual que `succ` es la función sucesor, la función predecesor es

$$\texttt{pred} \coloneqq \lambda n\, f\, x.\texttt{snd}(n(\lambda p.(\texttt{false}, \texttt{fst}\, p(\texttt{snd}\, p)(f(\texttt{snd}\, p))))(\texttt{true}, x)),$$

que cumple $\texttt{pred}\, 0 = 0$ y, para $n > 0$, $\texttt{pred}\, n = n - 1$.

Para iterar en cálculo λ se usan funciones recursivas, pero como las funciones no tienen nombre, hace falta algún mecanismo para que una función se llame a sí misma sin tener que usar su nombre.

Un **punto fijo** de una función f es una función F tal que $F = f\, F$. Llamamos **combinador paradójico**, **de punto fijo** u **operador de búsqueda del punto fijo** a un combinador $\mathbf{Y}$ tal que, para toda función f, $\mathbf{Y}\, f$ es un punto fijo de f. El primero lo encontró Curry,

$$\mathbf{Y} \coloneqq \lambda g.(\lambda x.g(x\, x))(\lambda x.g(x\, x)),$$

pues $\mathbf{Y} f = (\lambda x.f(x\, x))(\lambda x.f(x\, x)) = f((\lambda x.f(x\, x))(\lambda x.f(x\, x))) = f(\mathbf{Y} f)$. Con este podemos crear funciones recursivas, pues si $f = \lambda F\, x.E$, $\mathbf{Y}\, f = f(\mathbf{Y}\, f) = \lambda x.E[\mathbf{Y}\, f/F]$. Por ejemplo,

$$\texttt{factorial} \coloneqq \mathbf{Y}(\lambda f\, x.\texttt{iszero}\, x\, 1(x \cdot f(\texttt{pred}\, x))).$$

El nombre de combinador paradójico tiene que ver con la paradoja de Russell, en tanto que si $R \coloneqq \lambda x.\neg(x\, x)$ entonces $R\, R = \neg(R\, R)$ e $\mathbf{Y}(\lambda f\, x.\neg(f\, x))E$ no es normalizable para ningún E.

Escribimos

$$\texttt{let}\ x = E\ \texttt{in}\ T \coloneqq (\lambda x.T)E,$$

y permitimos las **ligaduras paralelas**

$$\texttt{let}\ x_1 = E_1\ \texttt{and}\ \cdots\ \texttt{and}\ x_n = E_n\ \texttt{in}\ T \coloneqq (\lambda(x_1, \ldots, x_n).T)(E_1, \ldots, E_n).$$

Una ligadura $x = \lambda b_1 \ldots b_n.E$ se puede sustituir por $x\, b_1 \ldots b_n = E$, $x = \mathbf{Y}(\lambda x.E)$ por $\texttt{rec}\ x = E$ y $x = \mathbf{Y}(\lambda x\, b_1 \ldots b_n.E)$ por $\texttt{rec}\ x\, b_1 \ldots b_n = E$, donde los b_i son símbolos o tuplas de símbolos. Finalmente, E `where` «bindings» significa `let` «bindings» `in` E.

Generalmente en un programa funcional se extraen expresiones comunes como construcciones `let`, con lo que un programa se mira como una serie de definiciones seguida por una expresión principal.

[1] Y lo hace pese a que no hemos definido las sustituciones con varias variables, qué curioso.

Capítulo 4

Haskell

Haskell es un lenguaje de programación funcional puro y perezoso.

4.1. Estructura léxica[1]

```
program =~ (<lexeme>|<whitespace>)*
lexeme =~ <qvarid>|<qconid>|<qvarsym>|<qconsym>|<literal>|\
          <special>|<reservedop>|<reservedid>
whitespace =~ <whitestuff>+
whitestuff =~ <whitechar>|<comment>|<ncomment>
whitechar =~ <newline>|[\v \t]|\p{Separator}
newline =~ \r\n|[\r\n\f]
comment =~ --+(?!<symbol>)<anynb>*<newline>
ncomment =~ \{-<anyseq>(<ncomment><anyseq>)*-\}
```

Las clases de caracteres son:

```
anyseq =~ (?!<any>*(\{-|-\})<any>*)<any>*
any =~ <graphic>|<whitechar>
anynb =~ <graphic>|[ \t]
graphic =~ <small>|<large>|<symbol>|<digit>|<special>|["']
small =~ \p{Lowercase_Letter}|_
large =~ \p{Uppercase_Letter}|\p{Titlecase_Letter}
symbol =~ [!#$%&*+./<=>?@\\^|~:-]|(?!<special>|[_"'])\p{Symbol}
digit =~ \p{Decimal_Digit_Number}
octit =~ [0-7]
hexit =~ <digit>|[A-Fa-f]
special =~ [()\[\]{},;`]
```

[1] Las reglas ***nombre =~ regex*** definen una expresión regular en formato PCRE ampliado con dos reglas: que ***<nombre>*** incluye la expresión regular nombrada entre paréntesis y que un \ antes de un salto de línea se salta todos los caracteres desde él mismo hasta el primer caracter de la siguiente línea que no es un espacio en blanco.

Las palabras reservadas son:

```
reservedid =~ case|class|data|default|deriving|do|else|foreign|\
              if|import|in|infix|infixl|infixr|instance|let|\
              module|newtype|of|then|type|where|_
reservedop =~ \.\.|:|::|=|\\|\||<-|->|@|~|=>
```

Las variables pueden estar en espacios de nombres aparentemente jerárquicos:

```
qvarid =~ (<conid>\.)*<varid>
qconid =~ (<conid>\.)*<conid>
qvarsym =~ (<conid>\.)*<varsym>
qconsym =~ (<conid>\.)*<consym>
```

Realmente sólo se distingue entre el nombre de la variable, detrás del último `.`, y el del módulo, antes del `.`, y no hay relación jerárquica entre módulos.

```
varid =~ (?!<reservedid>(?!<small>|<large>|<digit>|'))\
         <small>(<small>|<large>|<digit>|')*
conid =~ <large>(<small>|<large>|<digit>|')*
varsym =~ (?!((--+|<reservedop>)(?!<symbol>))|:)<symbol>+
consym =~ (?!<reservedop>(?!<symbol>)):<symbol>*
```

Los literales son:

```
literal =~ <integer>|<float>|<char>|<string>
integer =~ <digit>+|0[oO]<octit>+|0[xX]<hexit>+
float =~ <digit>+\.<digit>+(<exponent>)?|<digit>+<exponent>
exponent =~ [eE][+-]?<digit>+
char =~ '((?!['\\])<graphic>| |(?!\\&)<escape>)'
string =~ "((?!["\\])<graphic>| |<escape>|\\<whitechar>+\\)*"
escape =~ \\(<charesc>|<ascii>|<decimal>|o<octit>+|x<hexit>+)
charesc =~ [abfnrtv\\"'&]
ascii =~ \^[A-Z@\[\]\\^_]|NUL|SOH|STX|ETX|EOT|ENQ|ACK|BEL|BS|\
         HT|LF|VT|FF|CR|SO|SI|DLE|DC1|DC2|DC3|DC4|NAK|SYN|ETB|\
         CAN|EM|SUB|ESC|FS|GS|RS|US|SP|DEL
```

En principio un programa se interpreta como una secuencia de lexemas, `{` entra en un nuevo contexto y `}` sale del último contexto en que se ha entrado. Sin embargo, se pueden evitar apariciones de `{`, `}` y `;` sangrando los lexemas de cierta forma y dejando que el *lexer* los inserte como sigue:

1. Tras un `let`, `where`, `do` u `of` no seguido de `{`, o al principio del cuerpo si este no empieza por `module` o `{`, se inserta `{` y se abre un contexto implícito desde la columna por la que empieza el siguiente lexema (si lo hay). Este se cierra inmediatamente insertando `}` si está directamente dentro (sin ningún contexto «en medio») de un contexto implícito que empieza en una columna anterior.

2. Si el primer lexema explícito en una línea no es el primero de su contexto, mientras su contexto sea implícito, justo antes de la línea se inserta `;` si el lexema empieza en la misma columna que el contexto o `}` y si empieza en una anterior, cerrando el contexto.

3. Es un error cerrar un contexto implícito con un `}` explícito.

4. Mientras un lexema explícito en un contexto implícito cause un error sintáctico (los lexemas explícitos o implícitos anteriores forman un prefijo válido de la gramática libre de contexto pero al añadir el propio lexema ya no), se inserta `}` antes del lexema.

5. Al final de la entrada se insertan `}` suficientes para cerrar los contextos implícitos que queden abiertos.

4.2. Sesiones

El programa `ghci` lee expresiones o definiciones en Haskell, una por línea o entre una línea «`:{`» y una «`:}`», las evalúa e imprime el resultado, y también puede interpretar comandos de `ghci`. Una **sesión** es la secuencia de interacciones entre el usuario y este programa. Las definiciones también se pueden guardar en ficheros, generalmente con extensión `.hs`, llamadas **guiones** o ***scripts***, que se pueden cargar en el intérprete con el comando `:l` *`fichero`*. El formato es:[2]

```
module = body
body = "{" topdecls "}"
topdecls = [topdecl *(";" topdecl)]
topdecl = decl
```

Un **vínculo** es una asignación de un nombre a un valor, y un **entorno** o **contexto** es un conjunto de vínculos en que no hay más de un vínculo para el mismo nombre. Las expresiones se evalúan en un cierto contexto, y describen un valor que depende de este.

```
var = varid / "(" varsym ")"
varop = varsym / "`" varid "`"
con = conid / "(" consym ")"
conop = consym / "`" conid "`"
qvar = qvarid / "(" qvarsym ")"
qvarop = qvarsym / "`" qvarid "`"
qcon = qconid / "(" gconsym ")"
qconop = gconsym / "`" qconid "`"
gconsym = qconsym
cname = var / con
qop = qvarop / qconop
```

Las variables se nombran como `varid` y los **operadores** infijos como `varsym`, pero es posible usar un operador como nombre normal poniéndolo entre paréntesis y una función con nombre normal como operador poniéndola entre comillas invertidas. Se pueden definir nuevos operadores.

[2] Las reglas *`nombre = valor`* describen la gramática libre de contexto de Haskell en ABNF, con la extensión de que **`!`***`x`* indica que el siguiente elemento no debe tener forma *`x`*. Cada cadena entre comillas representa un sólo lexema y entre dos lexemas puede haber espacio. Esta gramática tiene ambigüedades que se resuelven haciendo las frases tan largas como sea posible de izquierda a derecha, resolviendo todos los conflictos *shift-reduce* con *shift*. Los *tokens* también se extienden lo máximo posible.

En el resto del capítulo cuando se menciona un nombre concreto, salvo que se indique lo contrario, se entiende que pertenece al módulo `Prelude`, el **preludio estándar**, cuyos vínculos están por defecto en el contexto global.

4.3. Tipos

Todo valor tiene un tipo con un cierto dominio, que incluye un **valor indefinido** $\perp$ correspondiente a las expresiones de ese tipo cuya evaluación no termina, aunque también se usa para notificar errores.

```
type = btype ["->" type]
btype = [btype] atype
atype = gtycon / varid / "(" type ")"
gtycon = qconid
```

Un tipo puede ser un `qconid`, cuyo significado depende del entorno, y esta definición puede aceptar un número fijo de parámetros de tipo.

El tipo *`fuente -> resultado`* es el de las funciones que toman **argumentos** del **tipo fuente** y devuelven **resultados** del **tipo resultado**. Una función f es **estricta** si $f \perp = \perp$. Dos funciones son **iguales** si devuelven los mismos resultados para los mismos argumentos, y entonces el compilador es libre de cambiar una por otra. Normalmente las funciones están currificadas, pues esto reduce el número de paréntesis y permite aplicar una función de varios argumentos a menos argumentos para obtener otra función que puede ser útil por sí misma. Es idiomático η-reducir las definiciones de funciones.

Un `varid` representa una variable libre, y cuando el tipo que se asigna a una expresión tiene alguna variable libre, el tipo de la expresión es **polimórfico**, genérico sobre los valores de las variables libres.

Tipos primitivos en el `Prelude`:

`Char` Caracteres Unicode.

`Double` Números de punto flotante de precisión doble.

`Float` Números de punto flotante de precisión simple.

`Int = minBound ... -1 | 0 | 1 ... maxBound` Enteros de tamaño limitado, con al menos los del rango $[-2^{29}, 2^{29})$.

`Integer = ... -1 | 0 | 1 ...` Enteros de tamaño arbitrario.

También son predefinidos las **tuplas**, productos directos de un número fijo de tipos escritos entre paréntesis, las **listas**, secuencias finitas de elementos del mismo tipo, escrito entre corchetes, y el **tipo unidad** `()`, con un solo elemento:

```
atype /= "(" type 1*("," type) ")" / "[" type "]"
gtycon /= "(" ")" / "[" "]"
```

Podemos definir nuevos tipos para aumentar la seguridad de los programas:

```
topdecl /= "data" [context "=>"] simpletype "=" constrs
           [deriving]
constrs = constr *("|" constr)
constr = conid *atype
       / (btype / "!" atype) conop (btype / "!" atype)
simpletype = conid *varid
```

Cada `constr` se refiere al tipo producto de los `atype` o `btype` etiquetado con el `conid` o `conop`, o bien a un tipo unipuntual con nombre `conid`, y el tipo `simpletype` es la unión disjunta de estos tipos producto y de $\{\bot\}$. Si `simpletype` contiene variables, estas corresponden a parámetros de tipo, y los `constrs` solo pueden contener variables que aparezcan en el `simpletype`. El `Prelude` define `data Bool = False | True`.

También es posible añadir nombres a los campos:

```
constr /= con "{" [fielddecl *("," fielddecl)] "}"
fielddecl = vars "::" (type / "!" atype)
vars = var *("," var)
```

Cada `var` define un campo del tipo producto, y por cada una se crea una función del tipo definido al tipo de la variable, que devuelve el valor en esa posición si la variable es de la variante `con` o $\bot$ en otro caso. Un **tipo recursivo** es uno que se tiene a sí mismo en la parte derecha de la definición, consiguiendo valores recursivos.

Los **sinónimos de tipo** permiten definir abreviaturas para tipos comunes, que internamente se expanden al tipo a la derecha:

```
topdecl /= "type" simpletype "=" type
```

Las cadenas de caracteres se definen como `type String = [Char]`.

```
topdecl /= "newtype" [context "=>"] simpletype = newconstr [
   deriving]
newconstr = con atype / con "{" var "::" type "}"
```

Crea un tipo que etiqueta a otro. Equivale a cambiar `newtype` por `data` pero sin añadir un $\bot$ adicional.

4.4. Clases de tipos

Son una forma de polimorfismo basada en clases cuyas instancias son tipos:

```
topdecl /= "class" [scontext "=>"] conid varid ["where" cdecls]
         / "instance" [scontext "=>"] qconid inst ["where" idecls]
cdecls = "{" [cdecl *(";" cdecl) "}"
inst = gtycon / "(" gtycon *varid ")"
     / "(" varid 1*("," varid) ")"
     / "[" varid "]" / "(" varid "->" varid ")"
idecls = "{" [idecl *(";" idecl)] "}"
```

Una declaración `class` define una clase `conid` y una serie de vínculos en el contexto global dados por las asignaciones de tipo en las `cdecl`, en las que `varid` se refiere a un tipo cualquiera de clase `conid` y debe aparecer alguna vez en el tipo asignado. Una **declaración de instancia** `instance` indica que el tipo `inst` es de la clase `qconid` y da las definiciones, llamadas **métodos**, de los vínculos establecidos por la clase cuando el tipo `varid` se sustituye por `inst`. Los nombres de métodos son miembros de una única clase, lo que evita conflictos entre nombres.

La declaración `class` puede contener definiciones por defecto para los vínculos, en cuyo caso no es necesario dar estas definiciones en las instancias, y también puede declarar la aridad de operadores definidos en la clase. Sin embargo, es común que las definiciones por defecto estén unas en función de otras y haya que definir al menos un elemento para romper el ciclo. La **definición mínima completa** de una clase es cualquier conjunto minimal de definiciones de la clase que deben definir las instancias.

Las declaraciones pueden tener un **contexto** para restringir los valores por los que se pueden sustituir las variables de tipo:

```
context = class / "(" [class *("," class)] ")"
class = qconid varid / qconid "(" varid 1*atype ")"
scontext = simpleclass / "(" [simpleclass *("," simpleclass)] ")"
simpleclass = qconid varid
```

Las sustituciones deben respetar que cada `varid` o `varid 1*atype` sea de la clase `qconid`. Esto es necesario cuando en la definición de una función se quieren usar funciones definidas en una clase, y de hecho el tipo de estas funciones en el entorno global tiene una restricción en su contexto.

Si una clase `class ... => Foo a` tiene `Bar a` en su contexto, `Bar` es **superclase** de `Foo` y `Foo` es **subclase** de `Bar`, y todo contexto que indique que una variable o expresión tiene tipo `Foo` permite usar sobre ella además de las operaciones de `Foo` las de `Bar`, recursivamente. El contexto de un método no puede tener restricciones sobre la variable de la clase.

Se pueden derivar automáticamente las definiciones de algunas clases predefinidas:

```
deriving = "deriving" (qconid / "(" [qconid *("," qconid)] ")"
```

Haskell soporta **tipos de alto rango**, lo que significa que las instancias de una clase pueden ser tipos que acepten parámetros, y no necesariamente tipos completos.

4.5. Expresiones

Haskell tiene **disciplina de tipos**, consistente en que toda expresión bien formada tiene un tipo deducible a partir de sus subexpresiones y los tipos de los vínculos en el contexto, y las expresiones a las que no se puede asignar un tipo están mal formadas. Esto permite detectar errores antes de la evaluación y fuerza al programador a plantearse tipos apropiados para los valores, ayudando a diseñar programas claros y bien estructurados.

Al empezar a evaluar una expresión, primero se comprueba que la sintaxis sea correcta y, si no lo es, se produce un **error de sintaxis**, y luego se comprueba si tiene un tipo correcto y si no se produce un **error de tipo**.

La **evaluación ansiosa** o **llamada por valor** es como el orden aplicativo pero sin evaluar en el interior de abstracciones, y la **evaluación perezosa** o **llamada por nombre** es como el

orden normal pero sin evaluar en el interior de abstracciones. Haskell usa evaluación perezosa, que al contrario que la ansiosa garantiza la terminación si la expresión es normalizable.

```
exp = infixexp ["::" type]
infixexp = lexp qop infixexp / "-" infixexp / lexp
lexp = fexp
fexp = [fexp] aexp / "let" decls "in" exp
aexp = qvar / literal / "(" exp ")"
qvar = qvarid / "(" qvarsym ")"
decls = "{" [decl *(";" decl)] "}"
```

Una variable (`qvar`) representa su valor en el contexto; un `literal` se representa a sí mismo; `fexp aexp` representa la aplicación de la función `fexp` al valor `aexp`; `lexp qop infixexp` equivale a `(qop) lexp infixexp`, aunque realmente distintos operadores tienen distinta precedencia del 0 al 9 y asociatividad por la izquierda o la derecha.

La negación `-`*`a`* equivale a `negate` *`a`*; los literales `char` son de tipo `Char`; los `string` de tipo `String`; los `integer` equivalen a `fromInteger (`*`x`*` :: Integer)`, siendo *`x`* el entero indicado, y los `float` a `fromRational (`*`x`*` :: Data.Ratio.Rational)`.

```
aexp /= "(" exp 1*("," exp) ")" / "[" exp *("," exp) "]" / gcon
gcon = "(" ")" / "[" "]" / qcon
gconsym /= ":"
```

La primera sintaxis indica una tupla, la segunda una lista de un tamaño dado, `()` el único elemento del tipo unidad, `[]` la lista vacía y un `qconid` es un constructor de tipo, que actúa como una función currificada que recibe tantos parámetros como aparezcan en su definición y del tipo correcto y devuelve un elemento del tipo correspondiente. Finalmente, `(:) :: a ->[a] ->[a]` es un operador asociativo por la derecha con precedencia 5 que añade un elemento al inicio de una lista, con lo que `[`*`a1`*`, ..., `*`an`*`]` equivale a *`a1`*`:...:`*`an`*`:[]`.

Una **sección** es un operador con una expresión delante o detrás:

```
aexp /= "(" infixexp qop ")" / "(" !"-" qop infixexp ")"
```

`(`*`infixexp qop`*`)` equivale a `\y ->` *`infixexp qop`* `y` y `(`*`qop infixexp`*`)` a `\x -> x` *`qop infixexp`*.

4.6. Patrones

```
pat = lpat / lpat qconop pat
lpat = apat / "-" (integer / float) / gcon 1*apat
apat = var ["@" apat] / literal / "_" / "(" pat ")" / gcon
     / qcon "{" [fpat *("," fpat)] "}" / "(" pat 1*("," pat) ")"
     / "[" pat *("," pat) "]"
fpat = qvar "=" pat
```

Un **patrón** recibe un argumento y bien encaja, creando una serie de vínculos, o no. El patrón `_` siempre encaja y no crea vínculos; una variable siempre encaja y crea un vínculo de la variable al argumento; un patrón de forma *`var`*`@`*`apat`* es similar pero solo encaja si lo hace *`apat`* y en tal caso crea un vínculo de *`var`* al argumento que se añade a los vínculos que

crea *apat*. Una misma variable no puede aparecer dos veces en la misma lista de patrones, ni siquiera como subpatrón. Un literal, su negación o un `gcon` suelto recibe un argumento del tipo que tendría como expresión, no crea vínculos y encaja sólo si el argumento es igual a él.

Un patrón `qcon 1*apat`, con tantos `apat` como parámetros tenga el constructor `qcon`, acepta un argumento del tipo que crearía el constructor y encaja si este es de la variante dada y cada `apat` encaja con el elemento correspondiente de la variante, añadiendo los vínculos de los `apat`. Uno `qcon { ... }` es similar pero para constructores definidos con llaves, y `( ... )` y `[ ... ]` hacen lo propio con tuplas y listas, entendiendo que las listas deben tener tantos elementos como patrones haya para encajar.

Un patrón no estándar es `varid "+" integer`, que encaja con un entero de valor mayor o igual al `integer` y asigna a la variable el entero menos ese `integer`.

Dada una lista de patrones o de listas de patrones del mismo tamaño cada una con una expresión asociada, el **encaje de patrones** consiste en seleccionar la expresión correspondiente al primer patrón que encaje con cierto argumento, o la primera lista en que cada patrón encaje con el correspondiente argumento de una lista de argumentos.

```
fexp /= "case" exp "of" "{" alts "}" /
     / "if" exp [";"] "then" exp [";"] "else" exp /
     / "\\" 1*apat "->" exp
alts = alt *(";" alt)
alt = pat ("->" exp / 1*gdpat) ["where" decls]
gdpat = guards "->" exp
guards = "|" guard
guard = infixexp
```

Si todas las `alt` son del segundo tipo, `case` *exp* `of` *alts* evalúa *exp*; selecciona la primera `gdpat` tal que *exp* encaja con el patrón de la `alt` en la que está y, en un contexto extendido por las declaraciones en `decls` y los vínculos de dicho patrón, la **guarda** (`guard`) devuelve `True`, y finalmente evalúa la expresión del `gdpat` en el contexto extendido y devuelve su valor, o bien devuelve $\perp$ si ninguna `gdpat` cumple la condición. Una `alt` de la forma `->` *exp* equivale a `| True ->` *exp*. Todas las guardas deben tener tipo `Bool` y todas las partes derechas de `gdpat` o `alt` de la misma `case` deben tener el mismo tipo o intersección no vacía, y el tipo de la expresión `case` será la intersección de los tipos de las partes derechas.

`if` *condition* `then` *when-true* `else` *when-false* equivale a `case` *condition* `of { True ->` *when-true* `; False ->` *when-false* `}`.

`\` $apat_1$ ... $apat_n$ `->`*exp* representa una función con parámetros definidos por patrones (generalmente variables) cuya aplicación a n parámetros encaja los parámetros si es posible y devuelve el valor de *exp* en un contexto extendido por los vínculos creados por los patrones, y devuelve $\perp$ en otro caso. El tipo de los parámetros puede ser restringido por los patrones y los usos que se les da a las variables en *exp*, y el tipo de la función es de la forma `[`*contexto* `=>]` t_1 `->` ... `->` t_n `->` r, donde los t_i son los tipos de los parámetros, que pueden contener variables posiblemente restringidas por el *contexto*, y r es el tipo que tiene el resultado cuando las variables tienen los tipos dados por los t_i. Los t_i y el contexto son los más generales posibles para que los patrones y la expresión sean legales, salvo que se asigne un tipo más restringido.

4.7. Otras expresiones

Una **secuencia aritmética** se define como:

```
aexp /= "[" exp ["," exp] ".." [exp] "]"
```

`[e1..]` equivale a `enumFrom e1`, `[e1,e2..]` a `enumFromThen e1 e2`, `[e1..e3]` a `enumFrom e1 e3` y `[e1,e2..e3]` a `enumFromThenTo e1 e2 e3`.

Para expresiones de tipo `Int`, `[x,y..]` es la secuencia aritmética infinita cuyos dos primeros elementos son x e y; `[x..]` equivale a `[x,x+1,..]`; `[x,y..z]` es como `[x,y..]` pero termina cuando se sobrepasa z, de modo que si $x \leq y$ la secuencia para en el primer elemento mayor a z sin incluirlo, y si $x > y$ la secuencia para en el primero menor que z sin incluirlo, y `[x..z]` equivale a `[x,x+1..z]`.

Una **lista por comprehensión** se define como

```
aexp /= "[" exp "|" qual *("," qual) "]"
qual = pat "<-" exp | exp
```

El resultado es una lista de elementos del tipo de `exp`. Un **cualificador** (`qual`) del primer tipo es un **generador** y uno del segundo es un **filtro** o **guarda**. Si e y f son expresiones, p un patrón y Q una lista de cualificadores, `[e | p <- f, Q]` evalúa, para cada elemento de la lista f que encaje con p, `[e | Q]`, en un entorno extendido por los vínculos del encaje, y concatena los resultados, y `[e | f, Q]` devuelve `[e | Q]` si el booleano f es `True` o `[]` en otro caso. Si Q es vacío se entiende que `[e | Q]` vale `[e]`.

```
lexp /= "do" "{" stmts "}"
stmts = *stmt exp [";"]
stmt = (exp / pat "<-" exp /) ";"
```

`do { exp }` equivale a *exp*, que debe tener tipo `m a` para algún `m` de clase `Monad` y algún `a`; `do { exp; rest }` equivale a `exp >> do { rest }`, y `do { pat <- exp; rest }` equivale a `exp >>= \pat -> do { rest }`.

4.8. Definiciones

Una **definición** añade un vínculo al contexto en que se encuentra, en principio el contexto global. Está formada por una **asignación de tipo** opcional, que indica el tipo de la variable, y una serie de ecuaciones. La asignación del tipo debe corresponder o al tipo inferido o a una restricción. No se puede dar más de una asignación a la misma variable y, si se asigna un tipo más restringido, no se puede usar la variable como si tuviera el tipo más general.

```
decl = gendecl / (funlhs / pat) rhs
cdecl = gendecl / (funlhs / var) rhs
idecl = (funlhs / var) rhs /
gendecl = vars "::" type / fixity [integer] ops
ops = op *("," op)
fixity = "infixl" / "infixr" / "infix"
```

Las funciones se definen con una serie de **ecuaciones** de la forma `funlhs rhs`:

```
funlhs = var 1*apat / pat varop pat
rhs = ("=" exp / 1*gdrhs) ["where" decls]
gdrhs = guards "=" exp
```

La **aridad** de una ecuación es el número de parámetros (número de `apat` en la primera sintaxis o 2 en la segunda), y todas las ecuaciones para la misma función `var` tienen la misma aridad. La función se define currificada, y para evaluarla sobre ciertos argumentos, se selecciona una **cláusula** (`gdrhs`) de la misma forma que se seleccionaban los `gdpat` en una expresión `case`, usando ecuaciones en vez de `alt`s y encajando tantos argumentos como sea la aridad, y se ejecuta la expresión correspondiente. Una ecuación *var apat*... `=` *exp* equivale a *var apat*... `|` `True` `=` *exp*. La inferencia de tipos funciona como si la función se hubiera definido mediante una abstracción con dentro un `case`.

Las declaraciones de la forma `pat rhs` definen variables globales añadiendo al contexto global los vínculos establecidos al encajar el `rhs` con el patrón, y es un error si no hay encaje. Generalmente `pat` es una variable.

El segundo tipo de `gendecl` establece que los operadores indicados tengan precedencia dada por el entero opcional y asociatividad por la izquierda (`infixl`), por la derecha (`infixr`) o ninguna (`infix`).

4.9. Módulos

```
module /= "module" qconid [exports] "where" body
exports = "(" [export *("," export) [","] ")"
export = qvar / qconid ["(" ".." ")"
       / "(" [cname *("," cname)] ")"]
       / qconid ["(" ".." ")" / "(" [qvar *("," qvar)] ")"]
       / "module" qconid
cname = var / con
```

Haskell organiza los elementos en módulos, uno por fichero, cada uno con un nombre único y un contexto global formado por los vínculos establecidos en el módulo y los que se importan de otros.

Una **entidad** es un vínculo importado o exportado por un módulo, con un nombre (que no incluye el nombre del módulo) y un valor.

Se exportan las entidades indicadas por `exports`: *qvar* para una variable, *qconid* para un tipo o clase, *qconid*`(..)` para el tipo y todos sus constructores o la clase y todos sus métodos; *qconid*`(`*cname*`, ...)` para el tipo y los constructores indicados, y *qconid*`(`*qvar*`, ...)` para el tipo y los métodos indicados. Si no hay un `exports`, se exportan todos los vínculos definidos en el módulo.

```
body /= "{" impdecls [";" topdecls] "}"
impdecls = impdecl *(";" impdecl)
impdecl = "import" ["qualified"] qconid ["as" qconid] [impspec]
impspec = ["hiding"] "(" [import *("," import)] ")"
import = var / qconid["(" ".." ")" / "(" [cname *("," cname)] ")"
       / qconid["(" ".." ")" / "(" [var *("," var)] ")"
```

Una `impdecl` importa entidades del módulo `qconid`, por defecto todas las que este importa, pero `impspec` permite importar sólo los vínculos de los `import`, con la misma semántica que los `export`, o sólo los que no están en los `import` si se indica `hiding`. Añadir `qualified` hace que las entidades se importen con nombre *módulo*`.`*entidad*, donde el *módulo* es el nombre del módulo del que se importa o el que se indica detrás de `as` si este aparece.

Algunos programadores prefieren usar `qualified` en todos los `import`, haciendo explícito el origen de cada nombre y usando `as` si hace falta para acortar el nombre, mientras que otros prefieren nombres cortos y usan `qualified` o `hiding` según sea necesario para evitar conflictos de nombres.

Los módulos permiten crear tipos de datos abstractos, pues exportar un tipo sin constructores hace que este solo se pueda usar fuera del módulo a través de las operaciones exportadas, permitiendo cambiar la representación del tipo sin afectar a sus usuarios.

Haskell no especifica una relación entre nombres de módulo y de fichero, pero GHC busca el módulo *A1*`.`.....`.`*An* en *A1*`/.../`*An*`.hs` en el directorio raíz del programa (el directorio actual si se usa GHCi) y una serie de directorios predefinidos.

Un **preludio** es un módulo que se importa por defecto en los demás como si hubiera un `import` *Preludio*, salvo que haya un `impdecl` explícito para ese preludio. El único es `Prelude`, con **operaciones primitivas** incluidas en el evaluador, como las operaciones aritméticas básicas, y tipos y funciones de utilidad.

Capítulo 5

El preludio estándar

Haskell tiene un preludio estándar con tipos y funciones predefinidos.

```
infixr 9 ., !!
infixr 8 ^
infixl 7 *, /, `quot`, `rem`, `div`, `mod`
infixl 6 +, -
infixl 5 ++
infix 4 ==, /=, <, <=, >=, >, `elem`
infixr 3 &&
infixr 2 ||
```

Cuando se indica ... en código significa que no se puede definir en Haskell, y cuando se indica con ,,, significa que sería demasiado engorroso. Si se define que un tipo es de una clase, también lo es de sus superclases, y si no se da la definición de la instancia de superclase es porque no se puede definir en Haskell.

5.1. Clases derivables

Estas clases se pueden derivar en una instancia `data` con al menos un constructor o una `newtype` si todos los tipos en los parámetros de todos los constructores son instancias de la clase en algún contexto y no hay una declaración de instancia que haga conflicto.

`()`, `(a,b)`, `(a,b,c)` y `[a]` implementan `Eq` y `Ord` como se indica, las tres primeras implementan también `Bounded` y `()` implementa también `Enum`.

5.1.1. Eq

```
class Eq a where  {-# MINIMAL (==) | (/=) #-}
    (==), (/=) :: a -> a -> Bool

    x /= y = not (x == y)
    x == y = not (x /= y)
```

`a == b` si `a` y `b` tienen el mismo constructor y todos sus campos son iguales (==), salvo que si un campo de `a` o `b` es ⊥, `a == b` devuelve ⊥ si todos los campos anteriores al primer ⊥ son iguales, y si `a` o `b` es ⊥ el resultado es ⊥.

5.1.2. Ord

```
data Ordering = LT | EQ | GT
    deriving (Eq, Ord, Enum, Read, Show, Bounded)

class (Eq a) => Ord a where  {-# MINIMAL compare | (<=) #-}
    compare :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min :: a -> a -> a
    compare x y
        | x == y = EQ
        | x <= y = LT
        | True   = GT
    x <= y = compare x y /= GT
    x <  y = compare x y == LT
    x >= y = compare x y /= LT
    x >  y = compare x y == GT
    max x y = if x <= y then y else x
    min x y = if x <= y then x else y
```

`a <= b` si `a == b` según la definición por defecto, `a` tiene un constructor definido más a la izquierda que el de `b` o ambos tienen el mismo y existe un campo cuyo valor en `a` es menor (<) que en `b` y todos los anteriores son iguales, salvo que `(<=)` devuelve ⊥ en los mismos casos que `(==)`.

5.1.3. Bounded

Sólo si ningún constructor tiene campos o solo hay un constructor.

```
class Bounded a where
    minBound, maxBound :: a
```

En un enumerado, `minBound` es el elemento del primer constructor y `maxBound` el del último, y en otro caso `minBound` y `maxBound` son los elementos del único constructor con el `minBound` o el `maxBound` de cada campo.

5.1.4. Enum

Sólo si ningún constructor tiene campos.

```
class Enum a where  {-# MINIMAL toEnum, fromEnum #-}
    toEnum :: Int -> a
    fromEnum :: a -> Int
    succ, pred :: a -> a
```

```
    enumFrom :: a -> [a]
    enumFromThen, enumFromTo :: a -> a -> [a]
    enumFromThenTo :: a -> a -> a -> [a]

    succ x = toEnum (fromEnum x + 1)
    pred x = toEnum (fromEnum x - 1)
    enumFrom x = map toEnum [fromEnum x ..]
    enumFromTo x y = map toEnum [fromEnum x .. fromEnum y]
    enumFromThen x y = map toEnum [fromEnum x, fromEnum y ..]
    enumFromThenTo x y z =
        map toEnum [fromEnum x, fromEnum y .. fromEnum z]
```

`toEnum 0` devuelve el primer elemento del enumerado (el del primer constructor), `toEnum 1` el segundo, etc., y para cualquier otro x, `toEnum` x devuelve $\bot$. `fromEnum` es la inversa de `toEnum`. Si `firstCon` es el primer constructor y `lastCon` el último, se define

```
enumFrom x = enumFromTo x lastCon
enumFromThen x y = enumFromThenTo x y bound
    where bound | fromEnum x <= fromEnum y = lastCon
                | otherwise                = firstCon
```

5.1.5. Show

```
type ShowS = String -> String

class Show a where  {-# MINIMAL showsPrec | show #-}
    showsPrec :: Int -> a -> ShowS
    show :: a -> String
    showsPrec _ x s = show x ++ s
    show x = showsPrec 0 x ""
```

`showsPrec` d x r devuelve una representación de x como expresión Haskell, entre paréntesis si el constructor de x tiene precedencia menor que d considerando que la precedencia de una aplicación funcional es 10, seguida de r.

La representación está formada por el nombre del constructor y las representaciones de los parámetros según su `showsPrec`, posiblemente separados por espacios, y añadiendo paréntesis si hacen falta por precedencia aunque no hagan falta por asociatividad. El constructor se muestra en forma infija si y sólo si es un `qconop`, y si usa sintaxis con llaves, la representación también, con los campos en el mismo orden. Puede haber más espaciado o paréntesis de lo necesario.

5.1.6. Read

```
type ReadS a = String -> [(a, String)]

class Read a where
    readsPrec :: Int -> ReadS a
```

`readsPrec d s` intenta leer del principio de *s* un elemento de tipo *a* devuelto por `showsPrec` y devuelve una lista de pares con el valor leído y el resto de la cadena, que será vacía si la entrada no es correcta. `(x,"")` debe ser un elemento de `readsPrec d (showsPrec d x "")`.

5.2. Booleanos

```
data Bool = False | True deriving (Eq, Ord, Enum, Read, Show,
   Bounded)

(&&), (||) :: Bool -> Bool -> Bool
True  && x = x
False && _ = False
True  || _ = True
False || x = x

not :: Bool -> Bool
not True  = False
not False = True

otherwise :: Bool
otherwise = True
```

5.3. Funciones

```
id :: a -> a
id x = x

const :: a -> b -> a
const x _ = x

(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)

flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

`until` *f p* evalúa *f* sobre un argumento hasta que cumple *p*.

```
until :: (a -> Bool) -> (a -> a) -> a -> a
until p f x | p x       = x
            | otherwise = until p f (f x)
```

`error` *s* devuelve $\perp$ y, cuando se evalúa, imprime el error *s*.

```
error :: String -> a
error s = ...
```

```
undefined :: a
undefined = error "Prelude.undefined"
```

5.4. Tuplas

```
fst :: (a, b) -> a
fst (x, y) = x

snd :: (a, b) -> b
snd (x, y) = y

curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)

uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x, y) = f x y
```

5.5. Clases numéricas

```
class (Eq a, Show a) => Num a where
        {-# MINIMAL (+), (*), abs, signum, fromInteger, #
          #          (negate | (-))                     #-}
    (+), (-), (*) :: a -> a -> a
    negate :: a -> a
    abs, signum :: a -> a
    fromInteger :: Integer -> a
    x - y = x + negate y
    negate x = 0 - x

class (Num a, Ord a) => Real a where
    toRational :: a -> Rational

class (Real a, Enum a) => Integral a where
        {-# MINIMAL quotRem, toInteger #-}
    quot, rem, div, mod :: a -> a -> a
    quotRem, divMod :: a -> a -> (a,a)
    toInteger :: a -> Integer
    divMod n d = let (q, r) = quotRem n d in
                   if signum r == - signum d then (q-1, r+d)
                   else (q, r)
    n `quot` d = q where (q, r) = quotRem n d
    n `rem` d = r where (q, r) = quotRem n d
```

```
    n `div` d = q where (q, r) = divMod n d
    n `mod` d = r where (q, r) = divMod n d

class (Num a) => Fractional a where
    (/) :: a -> a -> a
    fromRational :: Rational -> a

class (Fractional a) => Floating a where
    {-# MINIMAL pi, exp, log, sin, cos, asin, acos, atan, #
      #         sinh, cosh, asinh, acosh, atanh           #-}
    pi :: a
    exp, log, sqrt :: a -> a
    sin, cos, tan, asin, acos, atan :: a -> a
    sinh, cosh, tanh, asinh, acosh, atanh :: a -> a

    sqrt x = x ** 0.5
    tan x = sin x / cos x
    tanh x = sinh x / cosh x


class RealFrac a where  {-# MINIMAL properFraction #-}
        properFraction :: (Integral b) => a -> (b, a)
        truncate :: (Integral b) => a -> b

        truncate = fst . properFraction
```

5.6. Funciones numéricas

```
even, odd :: (Integral a) => a -> Bool
even n = n `rem` 2 == 0
odd = not . even

gcd, lcm :: (Integral a) => a -> a -> a
gcd x y = gcd' (abs x) (abs y)
          where gcd' x 0 = x
                gcd' x y = gcd' y (x `rem` y)
lcm _ 0 = 0
lcm 0 _ = 0
lcm x y = abs ((x `quot` (gcd x y)) * y)

(^) :: (Num a, Integral b) => a -> b -> a
x ^ 0     = 1
x ^ (n+1) = x * x^n

fromIntegral :: (Integral a, Num b) => a -> b
```

```
fromIntegral = fromInteger . toInteger

instance Bounded Int where ...
instance Integral Int where ...
instance Integral Integer where ...
instance RealFrac Float where ...
instance Floating Float where ...
instance RealFrac Double where ...
instance Floating Double where ...
```

5.7. Caracteres

```
instance Enum Char where ...
instance Bounded Char where
        minBound = '\0'
        maxBound = ...
instance Eq Char where
        c == c' = fromEnum c == fromEnum c'
instance Ord Char where
        c <= c' = fromEnum c <= fromEnum c'

type String = [Char]
```

5.8. Lectura y escritura

```
instance Show Int where ,,,
instance Read Int where ,,,
instance Show Integer where ,,,
instance Read Integer where ,,,
instance Show Float where ,,,
instance Read Float where ,,,
instance Show Double where ,,,
instance Read Double where ,,,
instance Show () where show _ = "()"
instance Read () where ,,,
instance Show Char where ,,,
instance Read Char where ,,,
instance (Show a) => Show [a] where showsPrec _ = showList
instance (Read a) => Read [a] where readsPrec _ = readList
instance (Show a, Show b) => Show (a, b) where ,,,
instance (Read a, Read b) => Read (a, b) where ,,,
```

`Int` e `Integer` se muestran como `<integer>` en decimal, `Float` y `Double` como `<float>` y `Char` como `<char>` en `showsPrec` y `readsPrec` y como `<string>` en `showList` y `readList`.

Capítulo 6

Listas y árboles

6.1. Operaciones sobre listas en `Prelude`

`map` aplica una función a todos los elementos de una lista y `filter` selecciona los que cumplen una condición.

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs
```

`(++)` concatena dos listas.

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

`length` devuelve la longitud de una lista.

```
length :: [a] -> Int
length [] = 0
length (_:l) = 1 + length l

head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs

last :: [a] -> a
last [x]    = x
```

```
last (_:xs) = last xs

init :: [a] -> [a]
init [x]    = []
init (x:xs) = x : init xs

null :: [a] -> Bool
null []    = True
null (_:_) = False
```

`take` ***n xs*** y `drop` ***n xs*** devuelven respectivamente los ***n*** primeros elementos y el resto de elementos de ***xs***.

```
take, drop :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ []         = []
take n (x:xs) = x : take (n-1) xs
drop n xs | n <= 0 = xs
drop _ []          = []
drop n (_:xs)  = drop (n-1) xs

takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p (x:xs) | p x = x : takeWhile p xs
takeWhile _ _            = []
dropWhile p []       = []
dropWhile p (x:xs)
        | p x        = dropWhile p xs
        | otherwise = xs
```

`(!!)` ***n xs*** devuelve el ***n***-ésimo elemento de ***xs***, empezando por el 0, pero es $O(n)$.

```
(!!) :: [a] -> Int -> a
(x:_) !! 0      = x
(_:xs) !! n = xs !! (n-1)
```

`zip` junta dos listas en una cuya longitud es la mínima de la de las dos listas y cuyos elementos son las tuplas de elementos correspondientes de ambas listas. `unzip` hace lo contrario.

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip _ = []

unzip :: [(a,b)] -> ([a], [b])
unzip xs = (map fst xs, map snd xs)
```

`foldl` ***f a xs*** aplica la función ***f*** de dos parámetros a ***a*** y al primer elemento de ***xs***, luego al resultado y al segundo elemento de ***xs***, etc., y devuelve el resultado final, y `foldr` ***f z xs*** aplica ***f*** al último elemento de ***xs*** y a ***z***, luego al penúltimo elemento y al resultado de esto, etc. (`foldl` empieza por la izquierda y `foldr` por la derecha).

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f a []     = a
foldl f a (x:xs) = foldl f (f a x) xs

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)

foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs

foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f xs = foldr f (last xs) (init xs)
```

`foldr` se suele usar para sumar una lista de elementos de un monoide, siendo `f` la suma y `z` el elemento neutro, y `foldr1` para hacer lo propio en un semigrupo.

`scanl`, `scanr`, `scanl1` y `scanr1` hacen lo mismo pero devolviendo una lista de resultados intermedios, de izquierda a derecha en `scanl` y `scanl1` y de derecha a izquierda en `scanr` y `scanr1`, tantos como la longitud que la lista de entrada en `scanl1` y `scanr1` y con un elemento más en `scanl` y `scanr`.

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f q xs = q : (case xs of [] -> []
                               x:xs -> scanl f (f q x) xs)

scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr f q0 [] = [q0]
scanr f q0 (x:xs) = let qs = scanr f q0 xs in f x (head qs) : qs

scanl1 :: (a -> a -> a) -> [a] -> [a]
scanl1 _ []     = []
scanl1 f (x:xs) = scanl f x xs

scanr1 :: (a -> a -> a) -> [a] -> [a]
scanr1 _ [] = []
scanr1 f xs = scanr f (last xs) (init xs)

iterate :: (a -> a) -> a -> a
iterate f x = x : iterate f (f x)
```

`concat` concatena una lista de listas.

```
concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss
```

Así, `[e | p <- xs, Q]` equivale a `concat (map f xs) where { f p = [e | Q]; f _ = [] }`, y `[e | c, Q]` equivale a `if c then [e | Q] else []`.

```
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []
```

```haskell
and, or :: [Bool] -> Bool
and = foldr (&&) True
or  = foldr (||) False

any, all :: (a -> Bool) -> [a] -> Bool
any p = or . map p
all p = and . map p

elem :: (Eq a) => a -> [a] -> Bool
elem x = any (==x)

sum, product :: (Num a) => [a] -> a
sum = foldl (+) 0
product = foldl (*) 1
```

6.2. Secuencias aritméticas en punto flotante

En `float` y `double` las secuencias aritméticas no se pueden definir como en el resto de tipos porque `toEnum` y `fromEnum` truncarían los números a enteros.

```haskell
instance Enum Float where
    toEnum = fromIntegral
    fromEnum = fromInteger . truncate
    succ x = x + 1
    pred x = x - 1
    enumFrom = iterate (+1)
    enumFromThen a a' = iterate (+(a'-a)) a
    enumFromThenTo a b = takeWhile (<= b+1/2) (enumFrom a)
    enumFromThenTo a a' b = takeWhile p (enumFromThen a a')
        where p | a' >= a   = (<= b + (a'-a)/2)
                | otherwise = (>= b + (a'-a)/2)
instance Enum Double where
        -- Igual que para Float
```

6.3. Árboles

Un **árbol binario** se define como

```haskell
data Árbol a = Hoja a | Rama (Árbol a) (Árbol a)

aplanar :: Árbol a -> [a]
aplanar (Hoja x) = [x]
aplanar (Rama xt yt) = aplanar xt ++ aplanar yt
```

El **tamaño** de un árbol es el número de hojas, y su **altura** es la distancia de la raíz a la hoja más lejana.

```
tam, altura :: Árbol a -> Int
tam = length . aplanar
altura (Hoja x) = 0
altura (Rama xt yt) = 1 + max (altura xt) (altura yt)

mapÁrbol :: (a -> b) -> Árbol a -> Árbol b
mapÁrbol f (Hoja x) = Hoja (f x)
mapÁrbol f (Rama xt yt) = Rama (mapÁrbol f xt) (mapÁrbol f yt)
```

Entonces `map f . aplanar = aplanar . mapÁrbol f`.

Un **árbol binario aumentado** es uno que tiene información en los nodos internos.

```
data ÁrbolAum r a = Hoja a | Rama r (ÁrbolAum r a) (ÁrbolAum r a)
type ÁrbolA = ÁrbolAum Int
```

`Árbol` equivale a `ÁrbolAum ()`. Un **árbol binario etiquetado** es uno en que la información no se guarda en las hojas sino entre dos subárboles.

```
data ÁrbolB a = Hoja | Rama (ÁrbolB a) a (ÁrbolB a)

aplanar :: ÁrbolB a -> [a]
aplanar Hoja = []
aplanar (Rama xt a yt) = aplanar xy ++ [a] ++ aplanar yt
```

Un **árbol binario de búsqueda** es un `Ord a => ÁrbolB a` para el que `aplanar` devuelve una lista en orden creciente. Es útil para operaciones de pertenencia, inserción y borrado.

Un **árbol binario montículo** es una instancia de

```
data ÁrbolM a = Hoja | Rama a (ÁrbolM a) (ÁrbolM a)
```

en que el valor de cada nodo es menor o igual al de cualquiera de los subárboles, y es útil para búsqueda del valor más pequeño o fusión de montículos. `ÁrbolM a` equivale a `ÁrbolAum a ()`.

Una **rosadelfa** es un árbol sin restricciones:

```
data Rosa a = Nodo a [Rosa a]
```

Una rosadelfa es k**-aria** si todo nodo tiene exactamente 0 o k subárboles, y los árboles binarios se pueden ver como rosadelfas binarias.

Capítulo 7

Entrada y salida

```
class Monad m where  {#- MINIMAL (>>=), return -#}
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    return :: a -> m a
    fail :: String -> m a

    m >> k = m >>= \_ -> k
    fail s = error s
```

Una **mónada** es una instancia de `Monad` con las siguientes propiedades:[1]

1. Identidad por la izquierda: `return a >>= h = ha`.
2. Identidad por la derecha: `m >>= return = m`.
3. Asociatividad: `(m >>= g) >>= h = m >>= (\x -> g x >>= h)`.

Una **mónada de E/S** es una mónada `m` que además tiene operaciones `putChar :: Char -> m ()` y `getChar :: m Char`, como `IO`. `IO a` es el tipo de las acciones que pueden realizar operaciones de E/S y devuelven un resultado de tipo `a`.

```
data IO a = ...
instance Monad IO where ...
```

`return` a es una acción que no hace nada y devuelve a, y m `>>=` h es una que ejecuta m, le pasa el resultado a h y devuelve el resultado de ejecutar la acción devuelta por h, permitiendo secuenciar acciones.

[1] Una mónada en X es un monoide en la categoría de los endofuntores de X, con el producto reemplazado por la composición de endofuntores y el conjunto unidad por el endofuntor identidad (`https://stackoverflow.com/questions/3870088/a-monad-is-just-a-monoid-in-the-category-of-endofunctors-whats-the-problem#3870310`). Una mónada es un como un burrito (`https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/`, `https://blog.plover.com/prog/burritos.html`). No voy a explicar la intuición de qué es una mónada porque en el examen sólo entra la definición formal e IO, pero hay un tutorial muy bueno en la Wiki de Haskell (`https://wiki.haskell.org/All_About_Monads`) y una explicación resumida (`https://wiki.haskell.org/Monad`).

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())

mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as = sequence_ (map f as)
```

Podemos definir (aunque no está en el `Prelude`):[2]

```
done :: Monad m => m ()
done = return ()
```

`putChar` imprime un caracter, `getChar` lee uno de la entrada estándar y `getContents` lee toda la entrada estándar en una `String` de forma perezosa según se va necesitando.

```
putChar :: Char -> IO ()
putChar = ...

getChar :: IO Char
getChar = ...

getContents :: IOString
getContents = ...

putStr, putStrLn :: String -> IO ()
putStr s = mapM_ putChar s
putStrLn s = do putStr s
                putStr "\n"

print :: Show a => a -> IO ()
print x = putStrLn (show x)

getLine :: IO String
getLine = do c <- getChar
             if c == '\n' then return ""
             else do s <- getLine
                     return (c:s)
```

`interact` f desactiva el *buffering* de la entrada y salida estándar, le pasa la entrada estándar a f y va imprimiendo el valor devuelto por la función, evaluándolo y pidiendo caracteres de la entrada según sea necesario.

```
interact :: (String -> String) -> IO ()
interact f = do hSetBuffering stdin NoBuffering
                hSetBuffering stdout NoBuffering
                s <- getContents
                putStr (f s)
        where hSetBuffering = ...
```

[2]Está en `Control.Monad.Extra` y se llama `skip`.

`readFile` lee el contenido de un fichero de forma perezosa, `writeFile` escribe un fichero y `appendFile` añade contenido al final de uno.

```
type FilePath = String
readFile :: FilePath -> IO String
readFile = ...

writeFile, appendFile :: FilePath -> String -> IO ()
writeFile = ...
appendFile = ...
```

Un programa en Haskell es una colección de módulos de los que uno es el principal y tiene un vínculo `main :: IO ()`, que se ejecuta al iniciar el programa.