Análisis de Lenguajes de Programación

Ignacio Rimini

August 2025

Índice

1. Unidad 1 - Sintaxis.

1.1. Definición. Sintaxis.

La **sintaxis** describe la forma que van a tener los programas válidos del lenguaje. Permite distinguir entre secuencias de símbolos que pertenecen al lenguaje y las que no.

Existen dos tipos de sintaxis:

1. **Sintaxis concreta.**
   - Modela las **secuencias de caracteres** que son aceptadas como programas sintácticamente válidos.
   - Incluye información sobre la **representación textual:**
     - cómo se parentiza una expresión,
     - cómo se separan los elementos de una lista,
     - si los operadores son prefijos, infijos o postfijos,
     - reglas de precedencia y asociatividad.
   - Es la sintaxis que se especifica típicamente mediante **gramáticas formales** (por ejemplo, BNF o EBNF).
2. **Sintaxis abstracta.**
   - Modela la **estructura esencial** de los programas válidos.
   - Es independiente de la representación textual.
   - Se centra en los componentes importantes para la semántica del programa, dejando de lado detalles como paréntesis redundantes o la notación exacta de los operadores.
   - Suele representarse mediante **árboles de sintaxis abstracta (ASTs).**

   En resumen, la **sintaxis concreta** es para los humanos (escribir programas correctamente, con reglas claras de formato y notación) y la **sintaxis abstracta** es para la máquina/compilador (trabajar con la estructura esencial sin el *ruido* textual).

1.2. Ejemplos de sintaxis.

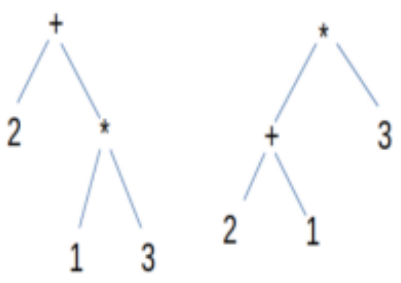
1.2.1. Ejemplo de sintaxis abstracta.

```
exp ::= n | exp + exp | exp * exp
```

donde $n \in \mathbb{N}$. Los siguientes son árboles sintácticos que corresponden a las expresiones:

$$2 + (1 * 3)$$

$$(2 + 1) * 3$$



1.2.2. Ejemplo de sintaxis concreta.

```
exp ::= n | exp '+' exp | exp '*' exp | '(' exp ')'
n ::= d | dn
d ::= '0' | '1' | ... | '9'
```

Y a continuación se describen algunas derivaciones:

- `exp` $\to$ `n` $\to$ `dn` $\to$ `dd` $\to$ `1d` $\to$ `12`
- `exp` $\to$ `exp '+' exp` $\to$ `exp '+' exp '*' exp` $\to$ `exp '+' exp '*' n` $\to \cdots \to$ `2 '+' 1 '*' 3`
- `exp` $\to$ `exp '*' exp` $\to$ `exp '+' exp '*' exp` $\to$ `exp '+' exp '*' n` $\to \cdots \to$ `2 '+' 1 '*' 3`

1.3. Gramáticas Libres de Contexto (CFG).

1.3.1. Introducción.

Una forma de definir la sintaxis de un lenguaje es mediante una **gramática libre de contexto (CFG).** La mayoría de lenguajes de programación definen su sintaxis mediante una CFG.

Un **lenguaje** resulta ser **libre de contexto** si hay una gramática libre de contexto que lo genera.

Las CFGs permiten capturar nociones esenciales de la sintaxis, como:

- Paréntesis balanceados.
- Palabras clave emparejadas (por ejemplo `begin` y `end`).

1.3.2. Definición. Grámatica Libre de Contexto.

Una **gramática libre de contexto (CFG)** puede ser definida por una 4-upla `(N, T, P, S)` donde:

- `N` es un conjunto finito de **no terminales** (categorías sintácticas o símbolos que no aparecen en el lenguaje: `exp, term, atom`, etc).
- `T` es un conjunto finito de **terminales** (símbolos básicos que si aparecen en el lenguaje: +, *, (,), números. identificadores, etc). Se cumple que `N` $\cap$ `T` $= \emptyset$.
- `P` es un conjunto de **producciones** de la forma $A \to \alpha$, donde $A \in N$ (es un no terminal) y $\alpha \in (N \cup T)^*$.

  Donde $*$ es el operador estrella de Kleene y el conjunto definido denota cualquier secuencia (incluyendo la secuencia vacía) de símbolos que sean terminales o no terminales.
- `S` es un **símbolo inicial** que pertenece a N.

1.3.3. Ejemplo de gramática libre de contexto.

Sea la gramática libre de contexto $G = (S, \{a, b\}, P, S)$ donde P tiene las siguientes reglas:

- $S \to aSb$
- $S \to \epsilon$

Esta gramática genera el lenguaje $\{a^n b^n : n \geq 0\}$.

Observaciones.

- Las producciones con el mismo lado izquierdo se pueden agrupar (notación de Backus-Naur o BNF): $S \to aSb \mid \epsilon$

- También se utiliza ::= en lugar de $\rightarrow$ en las producciones.
- El símbolo ϵ representa la cadena vacía.

1.3.4. Definición. Relación de derivación ($\Rightarrow$).

Sea $G = (N, T, P, S)$ una gramática libre de contexto, definimos la relación binaria:

$$\Rightarrow \subseteq (N \cup T)^* \times (N \cup T)^*$$

sobre secuencias de símbolos (terminales y no terminales), llamada **derivación** y donde $A \rightarrow B$ es una producción de G. Formalmente, $\Rightarrow$ es la menor relación tal que:

$$\alpha A \gamma \Rightarrow \alpha B \gamma$$

donde α y γ son secuencias de símbolos (terminales o no terminales), A es un no terminal que estamos reemplazando y B es un RHS (Right-Hand Side) de la producción.

Es decir, si tenemos una cadena donde aparece un no terminal A, podemos reemplazarlo por lo que dice la regla $A \rightarrow B$ manteniendo el contexto. Esta relación **derivación** es más abstracta que una regla de producción $\rightarrow$ de P, pues se puede definir para cualquier contexto.

1.3.5. Definición. Relación de derivación reflexiva-transitiva ($\Rightarrow^*$).

Dada la relación derivación $\Rightarrow$, se define la relación $\Rightarrow^*$ como la **clausura reflexiva-transitiva** de $\Rightarrow$. Es decir, es la menor relación sobre $(N \cup T)^*$ tal que:

- Si $\alpha \Rightarrow \beta$, entonces $\alpha \Rightarrow^* \beta$ (α deriva en muchos pasos a β).
- $\alpha \Rightarrow^* \alpha$
- Si $\alpha \Rightarrow^* \beta$ y $\beta \Rightarrow^* \gamma$, entonces $\alpha \Rightarrow^* \gamma$

1.3.6. Definición. Lenguaje generado por una gramática.

Dada una gramática G tenemos que un lenguaje L es generado por G si:

$$L(G) = \{w \mid w \in T^* \wedge S \Rightarrow^* w\}$$

Es decir, el lenguaje generado por G son todas las cadenas **formadas únicamente por terminales** que se pueden derivar desde el símbolo inicial S.

1.3.7. Ejemplo de derivación a partir de gramática.

Sea la gramática libre de contexto $G = (S, \{a, b\}, P, S)$ donde P tiene las siguientes reglas:

- $S \rightarrow aSb$
- $S \rightarrow \epsilon$

Tenemos que son válidas las siguientes derivaciones:

- $S \Rightarrow aSb$
- $S \Rightarrow \epsilon$
- $aSb \Rightarrow aaSbb$
- $aaS \Rightarrow aaaSb$ (aunque es imposible obtener el lado izquierdo con la gramática, la expresión vale)

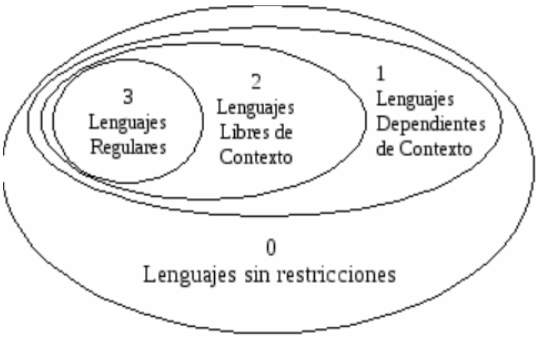
- $S \Rightarrow^* aSb$
- $S \Rightarrow^* \epsilon$
- $aSb \Rightarrow^* aaSbb$
- $aaSbb \Rightarrow^* aabb$
- $S \Rightarrow^* aaabbb$

Y luego, $L(G) = \{a^n b^n : n \geq 0\}$.

1.3.8. Propiedades de lenguajes libres de contexto.

- Los lenguajes libres de contexto pueden definirse también a través de autómatas no deterministas.
- La unión de dos lenguajes libres de contexto es también libre de contexto (la intersección no necesariamente).
- Determinar si dos CFGs generan el mismo lenguaje no es decidible.

A modo de observación, a continuación se inserta un gráfico sobre la **Jerarquía de Chomsky.**



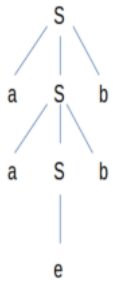
1.3.9. Definición. Árbol de parseo.

Un **árbol de parseo** es una derivación de una CFG $G = (N, T, P, S)$ si:

- Cada nodo del árbol tiene una etiqueta en $N \cup T \cup \{\epsilon\}$.
- La raíz tiene etiqueta S.
- Las etiquetas de los nodos interiores están en N.
- Si un nodo con etiqueta A tiene k hijos con etiquetas $X_1, ...X_k$, entonces $A \to X_1, ...X_k$ es una regla en P.
- Si un nodo tiene etiqueta ϵ, entonces es una hoja y es único hijo.

Luego, dada una gramática (N, T, P, S), decimos que $S \Rightarrow \alpha$ sii α es el **resultado** de un árbol de parseo, siendo éste la cadena que se forma al unir las etiquetas de las hojas del árbol (de izquierda a derecha).

Para la CFG del ejemplo, la cadena `aabb` tiene el siguiente árbol:



1.4. Gramáticas ambiguas.

1.4.1. Definición. Gramática ambigua.

Una gramática CFG G es **ambigua** si una palabra en `L(G)` tiene más de un árbol de parseo.

Observaciones.

- En general, los lenguajes CFG pueden ser generados por gramáticas ambiguas y no ambiguas. Pero existen algunos lenguajes que sólo pueden ser generados por CFG ambiguas. Estos son llamados **lenguajes inherentemente ambiguos.**
- Determinar si una CFG es ambigua no es decidible.

1.4.2. Desambiguar gramáticas.

Resolver la ambiguedad es importante, pues por ejemplo, dos árboles de parseo pueden tener distinta semántica.

Desambiguar fijando reglas de precedencia y asociatividad.

A continuación veremos un ejemplo de como desambiguar una gramática fijando reglas de precedencia y asociatividad entre los operadores. Tenemos la siguiente gramática (vista al inicio):

```
exp ::= n | exp '+' exp | exp '*' exp | '(' exp ')'
n ::= d | dn
d ::= '0' | '1' | ... | '9'
```

Podemos ver que hay dos derivaciones distintas para un mismo resultado. El resultado es $2 + 1 * 3$ y la primer derivación arranca con `exp '+' exp` y la segunda con `exp '*' exp`.

- `exp` $\to$ `exp '+' exp` $\to$ `exp '+' exp '*' exp` $\to$ `exp '+' exp '*' n` $\to \cdots \to$ `2 '+' 1 '*' 3`
- `exp` $\to$ `exp '*' exp` $\to$ `exp '+' exp '*' exp` $\to$ `exp '+' exp '*' n` $\to \cdots \to$ `2 '+' 1 '*' 3`

Para desambiguar la gramática, seguimos estos puntos que consisten en separar la gramática en niveles de precedencia. Así los no terminales sirven como *capas*: `exp` (suma y resta, nivel más bajo de precedencia), `term` (multiplicación y división, nivel intermedio), `atom` (paréntesis o números, nivel más alto).

- Definimos convenciones sobre la precedencia y asociatividad de los operadores. En este caso, reflejamos que el producto tiene más precedencia que la suma:

  ```
  exp ::= exp '+' exp | term
  term ::= term '*' term | atom
  atom ::= '(' exp ')' | n
  ```

Esto fuerza que siempre se reduzcan primero las multiplicaciones antes de llegar a las sumas: el producto tiene mayor precedencia que la suma.

- Reflejamos además que ambos operadores asocian a izquierda:

```
exp ::= exp '+' term | term
term ::= term '*' atom | atom
atom ::= '(' exp ')' | n
```

¿Por qué cambia? Antes: `exp ::= exp '+' exp` permitía que a la derecha hubiera otra `exp` completa, lo que da lugar a asociaciones a derecha.

Ahora: `exp ::= exp '+' term` fuerza a que lo que quede a la derecha sea un `term` (no una nueva suma completa), asegurando que las sumas se agrupan a izquierda.

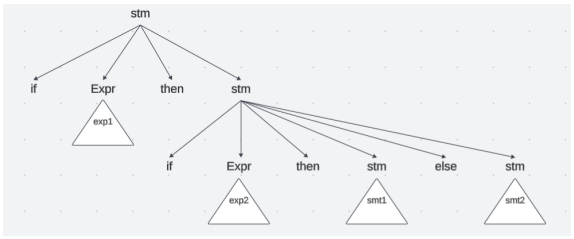
Problema de ambiguedad: *else colgado.*

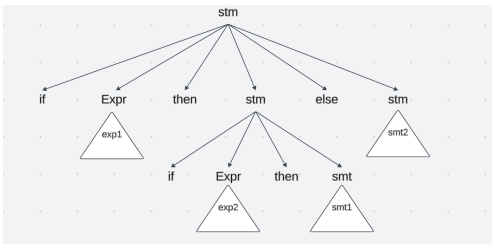
Supongamos una gramática con la siguiente regla de producción:

`stm ::= if exp then stm | if exp then stm else stm`

El siguiente programa tiene dos árboles de parseo:

`if` $\texttt{exp}_1$ `then if` $\texttt{exp}_2$ `then` $\texttt{stm}_1$ `else` $\texttt{stm}_2$





En el primer árbol, el `else` está asociado al `then` más cercano (el más a la derecha), mientras que en el segundo árbol está asociado al primer `then`. Es decir: primer árbol `if` $\texttt{exp}_1$ `then (if` $\texttt{exp}_2$ `then` $\texttt{stm}_1$ `else` $\texttt{stm}_2$`)` y segundo árbol `if` $\texttt{exp}_1$ `then (if` $\texttt{exp}_2$ `then` $\texttt{stm}_1$`) else` $\texttt{stm}_2$

Para transformar la gramática a una equivalente pero sin ambiguedades, agregamos la siguiente regla:

'Cada `else` está asociado al `then` más cercano que no está asociado a otro `else`'

Así, la sentencia que está entre `then` y `else` (matched) no puede contener un `if_then`, ya que esto violaría la regla.

```
stm -> matched | unMatched
matched -> if exp then matched else unMatched | otraSentencia
unMatched -> if exp then stm | if exp then matched else unMatched
```

Con esta reformulación todo `else` queda asociado de manera obligatoria al `then` más cercano. La derivación que asociaba un `else` a un `if` más externo deja de ser posible, eliminando la ambigüedad.

1.5. Árboles de sintaxis abstracta.

1.5.1. Introducción.

Hemos visto que al desambiguar una gramática concreta obtuvimos otra más díficil de interpretar. La **sintaxis abstracta** no tiene problemas de ambiguedad, porque se enfoca en la estructura jerárquica de los programas en lugar de su representación textual exacta: la sintaxis abstracta define árboles.

Un **Árbol de Sintaxis Abstracta (AST)** es una representación en forma de árbol de la estructura sintáctica de un lenguaje.

Las gramáticas abstractas son más simples que las concretas, porque eliminan detalles de notación (como paréntesis innecesarios o reglas de precedencia).

Ejemplo. La sintaxis de la siguiente gramática es concreta. Además, la gramática es ambigua:

```
exp ::= v | exp '+' exp | exp '-' exp | exp '*' exp | '(' exp ')'
v ::= x | y | z
```

Luego, el AST en notación BNF (Backus-Naur) de esta gramática es:

```
exp -> n | exp + exp | exp - exp | exp * exp
```

1.5.2. Implementación de AST.

Podemos implementar el AST de un lenguaje en Haskell con un tipo de datos algebraico. Por ejemplo:

```haskell
data Exp = Num Int | Sum Exp Exp | Prod Exp Exp | Minus Exp Exp
```

En la implementación notamos que los elementos de `Exp` son árboles y los nombres de los constructores son arbitrarios.

1.5.3. Lenguaje de booleanos y naturales.

Definimos el AST del lenguaje de booleanos y naturales como:

```
t -> true | false | if t then t else t | 0 | succ t | pred t | iszero t
```

Aquí `t` es una **metavariable**, que es una variable que pertenece al **metalenguaje** (lenguaje utilizado para analizar otro lenguaje) y es usada para representar un término del lenguaje objeto (el cual se describe).

1.5.4. Conjunto de términos.

Los AST definen el **conjunto de términos** del lenguaje. Una forma alternativa de definir el conjunto de términos de un lenguaje es mediante una definición inductiva.

Por ejemplo, definimos T como el menor conjunto tal que:

- $\{$`true, false, 0`$\} \subseteq T$
- Si $t \in T$ entonces $\{$`succ t, pred t, iszero t`$\} \subseteq T$
- Si $t, u, v \in T$ entonces `if t then u else v` $\in T$

Otra forma de definir el conjunto de términos (más concreta), es mediante un procedimiento que genera los elementos del conjunto. Por ejemplo, daremos una definición alternativa de T.

Primero damos una definición de S_i:

$$S_0 = \emptyset$$
$$S_{i+1} = \{\texttt{true, false, 0}\} \cup \{\texttt{succ t, pred t, iszero t} \mid \texttt{t} \in S_i\} \cup \{\texttt{if t then u else v} \mid \texttt{t, u, v} \in S_i\}$$

Y luego definimos:

$$S = \bigcup_{i \in N} S_i$$

Dado que T fue definido como el menor conjunto que satisface ciertas condiciones, para probar $T = S$, basta con probar:

1. S satisface las condiciones de T,
2. cualquier conjunto que satisfaga las condiciones contiene a S (S es el menor conjunto que satisface las condiciones)

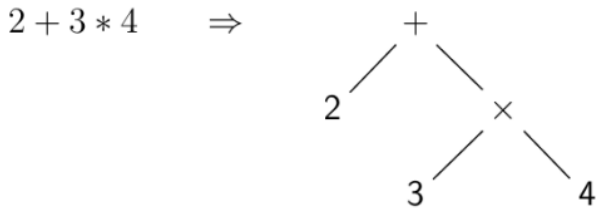
2. Unidad 2 - Analizadores Sintácticos (Parsers).

2.1. Introducción.

2.1.1. ¿Qué es un parser?

Un **parser** (o analizador sintáctico) es un programa que toma una cadena de caracteres como entrada y determina su **estructura sintáctica,** es decir, cómo se organiza de acuerdo con las reglas de una gramática formal.

El parser no se queda con el texto plano, sino que construye una representación estructurada (generalmente un árbol de sintaxis) que luego puede ser utilizado por otros componentes de un programa.



Se usan en la mayoría de los programas que necesitan **interpretar, transformar o validar entradas estructuradas.** Algunos ejemplos comunes son:

- **Calculadoras y lenguajes matemáticos.** Antes de evaluar una expresión como $3 + 4 * 2$, el parser construye un árbol de sintaxis que respeta la precedencia de los operadores.
- **Compiladores e intérpretes de lenguajes de programación.** Un compilador utiliza un parser para transformar el código fuente en una estructura intermedia que luego puede optimizarse y traducirse a lenguaje máquina.
- **Navegadores web.** Cuando un navegador recibe un documento HTML, primero lo parsea para construir el **DOM (Document Object Model),** que luego renderiza en pantalla.
- **Procesadores de datos.** Archivos como JSON, XML o CSV son parseados para convertirlos en estructuras de datos que los programas pueden manipular fácilmente.

2.1.2. El tipo de los parsers en Haskell.

Un parser en lenguajes funcionales (como Haskell) puede representarse con distintos tipos de funciones, dependiendo de qué información queremos capturar.

1. **Versión básica.** La idea más simple de parser es la siguiente, que toma una cadena y devuelve un árbol. Pero este tipo no es suficiente, porque no contempla si el parser consume toda le entrada ni qué sucede con lo que sobra.

```haskell
type Parser = String -> Tree
```

2. **Parser con resto de entrada.** Un parser debería devolver no solo el resultado, sino también la parte de la cadena que no consumió. Así podemos encadenar parsers de manera secuencial, reutilizando lo que queda de la entrada.

```haskell
type Parser = String -> (Tree, String)
```

3. **Parser con posibilidad de falla.** Un parser también puede **fallar.** Para capturar esto, se utiliza una lista de posibles resultados:

```haskell
type Parser = String -> [(Tree, String)]
```

- Lista vacía: falla (no se pudo parsear).
- Lista unitaria: éxito.
- Más de un resultado: posibles parseos alternativos (útil en gramáticas ambiguas).

4. **Generalización con tipos polimórficos.** No siempre el parser devuelve un árbol, a veces puede devolver cualquier tipo de valor. Por eso se define un tipo de dato polimórfico:

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

Esto permite tener parsers que devuelvan diferentes estructuras (números, expresiones, árboles, etc).

2.2. Parsers varios y operaciones.

2.2.1. Parsers básicos.

Los parsers más complejos se construyen a partir de **parsers básicos** que sirven como bloques fundamentales.

- **Parser que siempre tiene éxito.** No consume la entrada y devuelve un valor dado: devuelve siempre una lista unitaria con el valor `v` y la entrada intacta.

```haskell
return :: a -> Parser a
return v = \s -> [(v,s)]
```

- **Parser que siempre falla.** Representa un parser sin resultados: siempre devuelve la lista vacía, indicando que el parseo no tuvo éxito.

```haskell
failure :: Parser a
failure = \_ -> []
```

- **Parser que consume un caracter.** Devuelve el primer caracter de la entrada (junto con el resto), o falla si la cadena está vacía.

```haskell
item :: Parser Char
item (x:xs) = [(x, xs)]
item [] = []
```

- **Función auxiliar `parse`.** Sirve para aplicar un parser a una cadena.

```haskell
parse :: Parser a -> String -> [(a, String)]
parse p s = p s
```

Ejemplos.

```haskell
parse (return 1) "abc"
-- [(1, "abc")]

parse failure "abc"
-- []

parse item "abc"
-- [("a", "bc")]

parse item ""
-- []
```

2.2.2. Combinadores de parsers: CHOICE.

El combinador **choice** `<|>` pemite probar dos parsers en orden:

- Si el primero `p` tiene éxito, usamos su resultado.
- Si el primero falla, se prueba con el segundo (`q`).

```
(<|>) :: Parser a -> Parser a -> Parser a
(p <|> q) s = case p s of
                [] -> parse q s          -- Si p falla, usamos q
                [(v, out] -> [(v, out]   -- Si p tiene exito, nos quedamos con eso.
```

Ejemplos.

```
(item <|> return '1') "abc"
-- [('a', "bc")]

(item <|> return '1') ""
-- [('1', "")]
```

2.2.3. Combinadores de parsers: SECUENCIAMIENTO.

Muchas veces queremos combinar varios parsers en una misma operación. Por ejemplo, aplicar un parser después de otro y juntar sus resultados.

Se podría pensar en un combinador de tipo:

```
Parser a -> Parser b -> Parser (a, b)
```

Sin embargo, en la práctica es más general y conveniente usar el **operador de secuenciamiento:**

```
(>>=) :: Parser a -> (a -> Parser b) -> Parser b
(p >>= f) s = case parse p s of
                [] -> []
                [(v, out)] -> parse (f v) out
```

Este operador permite:

1. Ejecutar un parser `p`.
2. Usar el valor que devolvió (`v`).
3. Construir a partir de él un nuevo parser con `f v`.
4. Continuar el parseo sobre la entrada restante.

Uso de la notación `do`.

El encadenamiento con el operador $\gg=$ se puede escribir de forma más clara con **notación** `do` (sintaxis azucarada de Haskell para mónadas):

```
p1 >>= \v1 ->
p2 >>= \v2 ->
...
pn >>= \vn ->
return (f v1 v2 ... vn)
```

Se puede reescribir como:

```
do  v1 <- p1
    v2 <- p2
    ...
    vn <- pn
    return (f v1 v2 ... vn)
```

- Se aplica la **regla de layout:** cada parser debe comenzar en la misma columna.

- Si el valor de un parser intermedio no se usa, se puede omitir el ←:

```
do  p1
    v <- p2
    return v
```

- El valor devuelto por la secuencia es el valor del último parser.
- Esta notación no es exclusiva de los parsers, sino de cualquier mónada (listas, Maybe, IO, etc).

Ejemplo.

```
-- Ejemplo: aplica 3 parsers y devuelve el resultado del 1ero y 3ero en un par.
-- Sin utilizar notacion DO.
p :: Parser (Char, Char)
p = item >>= \x ->
    item >>= \_ ->
    item >>= \y ->
    return (x, y)

-- Utilizando notacion DO: hace falta declarar Parser como monada.
p :: Parser (Char, Char)
p = do  x <- item
        item
        y <- item
        return (x, y)

parse p "abcdef"
-- [('a', 'c'), "def"]

parse p "ab"
-- []
```

2.2.4. Primitivas derivadas.

Además de los parsers básicos, podemos definir **parsers más complejos** construidos a partir de los anteriores. Estos se llaman **primitivas derivadas.**

- **sat. Parser con predicado.** Parsea un caracter solo si satisface una condición booleana.

```
sat :: (Char -> Bool) -> Parser Char
sat p = do  x <- item
            if p x  then return x
                    else failure
```

- **Parser de dígitos.** Utilizando `sat` podemos parsear si un caracter en una cadena es un dígito.

```
digit :: Parser Char
digit = sat isDigit
```

- **Parser de caracter dado.** Utilizando `sat` podemos parsear si un caracter de una cadena es igual a un caracter específico.

```
char :: Char -> Parser Char
char x = sat (x ==)
```

- **Parser de una cadena dada.** Este parser intenta reconocer una cadena específica dentro de la entrada. La definición es:

```
string :: String -> Parser String
string [] = return []
string (x:xs) = do  char x
                    string xs
                    return (x:xs)
```

El caso base `string []` se activa cuando ya no quedan caracteres por parsear. Allí simplemente devolvemos un parser que no consume entrada y retorna la cadena vacía:

```
return []
-- que devuelve [([], restoEntrada)]
```

En el caso recursivo `string (x:xs)` se realizan tres pasos:

1. `char x`: verifica que el primer carácter de la entrada coincida con `x`. Si lo consume correctamente, continúa con el resto de la entrada.
2. `string xs`: llama recursivamente al parser para el resto de la cadena esperada.
3. `return (x:xs)`: al volver de la recursión, reconstruye la cadena original agregando el carácter actual al frente del resultado acumulado.

Ejemplo: Supongamos que queremos parsear la cadena `."bc"` sobre la entrada `."bcdef"`.

1. En el primer nivel, se aplica `char 'a'` sobre `'abcdef'`, lo cual devuelve `[('a', "bcdef")]`. Luego se llama recursivamente a `string "bc"`.
2. En el segundo nivel, se aplica `char 'b'` sobre `'bcdef'`, devolviendo `[('b', 'cdef')]`. Luego se llama recursivamente a `string 'c'`.
3. En el tercer nivel, se aplica `char 'c'` sobre `'cdef'`, devolviendo `[('c', 'def')]`. Ahora la recursión llama a `string []`.
4. El caso base `string []` devuelve `[([], 'def')]`.

A medida que la recursión se desarma, se reconstruye la cadena paso a paso:

- `string []` devuelve `[([], 'def')]`.
- La llamada con `'c'` produce `[('c', 'def')]`.
- La llamada con `'b'` produce `[('bc', 'def')]`.
- Finalmente, la llamada con `'a'` devuelve `[('abc', 'def')]`.

En conclusión, el parser `string 'abc'` sobre la entrada `'abcdef'` devuelve:

```
[("abc", "def")]
```

De esta manera, la cadena objetivo se reconstruye al volver de cada llamada recursiva, mientras que el resto de la entrada sin consumir se propaga hacia arriba en toda la evaluación.

- **Repetición de parsers.** Los parsers `many` y `many1` son mutuamente recursivos y aplican un parser muchas veces hasta que fallan, devolviendo los valores parseados en una lista.

```haskell
-- Aplica un parser 0 o mas veces.
many :: Parser a -> Parser [a]
many p = many1 p <|> return []

-- Aplica un parser 1 o mas veces.
many1 :: Parser a -> Parser [a]
many1 p = do    v <- p
                vs <- many p
                return (v:vs)
```

- **Parser de un número natural.** Parsea un número natural de una cadena dada, y lo devuelve directamente como número (no string).

```haskell
nat :: Parser Int
nat = do     xs <- many1 digit
             return (read xs)
```

- **Parser de espacios.** Parsea 0 o más espacios.

```haskell
space :: Parser ()
space = do  many (sat isSpace)
            return ()
```

2.2.5. Parser ignorando espacios.

La siguiente primita aplica un parser ignorando espacios:

```haskell
token :: Parser a -> Parser a
token p = do    space
                v <- p
                space
                return v
```

Con esta primitiva, tenemos los siguientes ejemplos:

- **Parsear una cadena ignorando espacios.**

```haskell
-- Parsea una cadena ignorando espacios.
symbol :: String -> Parser String
symbol xs = token (string xs))
```

- **Parsear un natural ignorando espacios.**

```haskell
-- Parsea un natural ignorando espacios.
natural :: String -> Parser Int
natural xs = token (nat xs)
```

Y podemos aplicarlas en el siguiente ejemplo, que parsea una lista de naturales.

```haskell
listnat :: Parser [Int]
listnat = do    symbol "["
                n <- natural
                ns <- many (do  symbol ","
                                natural)
                symbol "]"
                return (n:ns)

parse listnat "[1,2,3]"
-- [([1,2,3],"")]

parse listnat "[1,2,3] hola que tal"
-- [([1,2,3]," hola que tal")]

parse listnat "1,2,3 hola que tal"
-- []
```

2.3. Parser de gramáticas.

2.3.1. De gramática a parser en Haskell.

En esta sección veremos cómo transformar una gramática libre de contexto en un parser funcional escrito en Haskell. El proceso se puede dividir en varias etapas claras:

1. **Definir la gramática formal.** El primer paso es escribir la gramática que describe el lenguaje de expresiones aritméticas. Por ejemplo:

   ```
   exp -> term '+' exp | term
   term -> atom '*' term | atom
   atom -> '(' exp ')' | n
   n -> d | dn
   d -> '0' | '1' | ... | '9'
   ```

2. **Factorizar la gramática.** Muchas veces la gramática original introduce ambigüedad o backtracking innecesario. Para mejorar la eficiencia del parser, reescribimos las reglas:

   ```
   exp -> term ('+' exp | e)
   term -> atom ('*' term | e)
   ```

3. **Asociar cada no terminal con un parser.** Cada símbolo no-terminal de la gramática se traduce a una función parser en Haskell:

   - El parser `expr` corresponde al no-terminal `exp`.
   - El parser `term` corresponde al no-terminal `term`.
   - El parser `factor` corresponde al no-terminal `atom`.

4. **Implementar los parsers en Haskell.** Utilizando combinadores de parsers (`do`-notation, `<|>`, etc.), escribimos las definiciones:

   ```haskell
   expr :: Parser Int
   expr = do t <- term
             do symbol "+"
                e <- expr
                return (t+e)
             <|> return t

   term :: Parser Int
   term = do f <- factor
             do symbol "*"
                t <- term
                return (f*t)
             <|> return f

   factor :: Parser Int
   factor = do symbol "("
               e <- expr
               symbol ")"
               return e
            <|> natural
   ```

   - `expr` primero reconoce un `term`, y opcionalmente un `+ expr`.
   - `term` reconoce un `factor`, y opcionalmente un `* term`.
   - `factor` reconoce un número natural o una expresión entre paréntesis.

5. **Construir un evaluador.** Una vez que el parser puede reconocer la estructura de una cadena, podemos asociarle semántica (evaluar su valor numérico):

```
eval :: String -> Int
eval xs = fst (head (parse expr xs))
```

Por ejemplo: `eval "2*3+4"`→ `10` y `eval "2*(3+4)` → `14`.

2.3.2. Asociatividad de operadores y recursión a izquierda.

Para que el operador asocie a izquierda debemos modificar la gramática:

```
exp -> exp ('+' term | '-' term | e)
term -> term ('*' atom | '/' atom | e)
```

Pero esta gramática presenta el problema de la recursión a izquierda y por lo tanto el parser correspondiente (el que sigue) no terminará. La explicación es que siempre estamos evaluando `expr`, luego `expr` y así sucesivamente, sin consumir nada de la cadena ingresada.

```
1 expr :: Parser Int
2 expr =  do  t <- expr
3             char '+'
4             e <- term
5             return (t+e)
6         <|> term
```

Por lo tanto, transformamos la gramática en otra equivalente que no tenga recursión a izquierda (sino a derecha). Transformamos la siguiente expresión de la gramática:

$$A \to A\alpha \mid \beta$$

donde α es no vacío y β no empieza con A, en la siguiente gramática:

$$A \to \beta A'$$
$$A' \to \epsilon \mid \alpha A'$$

y luego definimos el nuevo parser para esta gramática.

3. Unidad 3.1 - Semántica.

3.1. Introducción.

3.1.1. Definición formal de un lenguaje.

La **sintaxis** especifica cómo se construyen los programas de un lenguaje (qué cadenas son válidas). En las últimas unidades vimos:

- Cómo generar un lenguaje libre de contexto (CF) a partir de una CFG.
- Cómo construir un analizador sintáctico (parser) que reconozca un lenguaje generado por una CFG.

La **semántica** da *significado* a cada programa gramaticalmente correcto del lenguaje: explica qué hace un programa cuando se ejecuta.

3.1.2. Beneficios de la semántica formal.

Una semántica bien definida permite:

- **Implementación:** sirve como especificación para intérpretes y compiladores. Además permiten garantizar que las optimizaciones son correctas.
- **Análisis estático:** dado que la semántica provee la base para razonar sobre programas, es necesaria para probar propiedades: propiedades de correctitud y propiedades de seguridad.
- **Diseño de lenguajes:** ayuda a resolver ambigüedades y a elegir decisiones de diseño (orden de evaluación, control de efectos). Además, el uso de las matemáticas pueden sugerir estilos de programación.

3.1.3. Enfoques clásicos de semántica.

- **Semántica operacional.** Captura el significado de un programa como una relación que describe cómo se ejecuta.
- **Semántica denotacional.** El significado de un programa es modelado por objetos matemáticos que representan el efecto de ejecutar las construcciones y pertenecen a un dominio $\mathcal{D}$.

  Se define la función de interpretación: $[\![\cdot]\!] : \mathcal{T} \to \mathcal{D}$
- **Semántica axiomática.** Especifica propiedades sobre el efecto de ejecutar programas. Enfoque basado en la lógica metemática, la más conocida es la lógica de Hoare usada para probar la correctitud de programas imperativos.

3.2. Tipos de semántica operacional.

La **semántica operacional** nos proporciona un modelo para la implementación de un intérprete o compilador del lenguaje. Se distinguen dos tipos de acuerdo a los detalles de ejecución que brindan:

1. **Semántica de paso chico (small-step):** la evaluación se describe **paso a paso** mediante una relación de reducción

   $$t \to t'$$

   Una ejecución es una cadena de pasos $t \to t_1 \to t_2 \to \cdots$. Es útil para especificar máquinas abstractas, concurrencia y efectos intermedios.
2. **Semántica de paso grande (big-step):** la evaluación relaciona directamente una expresión con su resultado final, ocultando cómo se llega al resultado:

   $$t \Downarrow v$$

No muestra pasos intermedios; es más compacta y adecuada para definir el resultado total de una evaluación.

Ambos tipos de semántica se definen mediante una **relación de transición.**

3.2.1. Ejemplo de semántica operacional big-step.

Definimos el conjunto de términos $\mathcal{T}$ como:

```
t -> T | F | if t then t else t
```

Y el conjunto de valores $\mathcal{V}$ ($\mathcal{V} \subseteq \mathcal{T}$):

```
v -> T | F
```

Los términos t y v son metavariables que denotan términos y valores respectivamente. La semántica **big-step** va a relacionar un término con un valor.

Definimos la relación 'reduce a': $\Downarrow \subseteq \mathcal{T} \times \mathcal{V}$ como la menor relación que satisface estas reglas:

$$\frac{}{v \Downarrow v} \quad \text{(B-Val)}$$

$$\frac{t_1 \Downarrow \texttt{T} \qquad t_2 \Downarrow v}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Downarrow v} \quad \text{(B-IfTrue)}$$

$$\frac{t_1 \Downarrow \texttt{F} \qquad t_3 \Downarrow v}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Downarrow v} \quad \text{(B-IfFalse)}$$

Observaciones.

- La primer regla es un axioma.
- Cada regla define un **esquema** de reglas, dado que t_1, t_2, t_3 y v son metavariables. A partir de ellas se deducen infinitas reglas, por ejemplo:

$$\frac{}{\texttt{T} \Downarrow \texttt{T}}$$

$$\frac{\texttt{T} \Downarrow \texttt{F} \qquad \texttt{F} \Downarrow \texttt{F}}{\texttt{if T then T else F} \Downarrow \texttt{F}}$$

- Cuando $(t, v) \in \Downarrow$, es decir (t, v) pertenece a la relación de evaluación decimos t **deriva a** v, y escribimos $t \Downarrow v$.
- Mostraremos que $t \Downarrow v$ mediante un **árbol de derivación,** cuyas hojas van a ser instancias de los axiomas y los nodos internos instancias de las reglas.

3.2.2. Ejemplo de árbol de derivación para semántica big-step.

Probaremos que `if (if F then T else T) then F else T` $\Downarrow$ `F` con un árbol de derivación:

$$\dfrac{\dfrac{\dfrac{}{\texttt{F} \Downarrow \texttt{F}}\,(\text{B-Val}) \quad \dfrac{}{\texttt{T} \Downarrow \texttt{T}}\,(\text{B-Val})}{\texttt{if F then T else T} \Downarrow \texttt{T}}\,(\text{B-IfFalse}) \quad \dfrac{}{\texttt{F} \Downarrow \texttt{F}}\,(\text{B-Val})}{\texttt{if (if F then T else T) then F else T} \Downarrow \texttt{F}}\,(\text{B-IfTrue})$$

3.2.3. Relación de evaluación small-step.

La semántica **small-step** se da por una relación entre *estados* de una máquina abstracta. Definimos la relación de evaluación $\rightarrow \subseteq \mathcal{T} \times \mathcal{T}$ con las siguientes reglas:

$$\texttt{if T then } t_2 \texttt{ else } t_3 \rightarrow t_2 \quad (\text{E-IfTrue})$$

$$\texttt{if F then } t_2 \texttt{ else } t_3 \rightarrow t_3 \quad (\text{E-IfFalse})$$

$$\dfrac{t_1 \rightarrow {t_1}'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \rightarrow \texttt{if } {t_1}' \texttt{ then } t_2 \texttt{ else } t_3}\,(\text{E-If})$$

Observaciones.

- $t \rightarrow t'$ se lee *t evalúa a t' en un solo paso.*
- Notar que `T` y `F` no evalúan a nada.
- Las reglas a veces se dividen en reglas de **computación** (`E-IfTrue` y `E-IfFalse`) y reglas de **congruencia** (`E-If`).
- La relación de evaluación fija una **estrategia de evaluación.** En `if t1 then t2 else t3`, se debe evaluar `t1` antes de evaluar `t2` o `t3`.

3.2.4. Ejemplo de árbol de derivación para semántica small-step.

Sean:

```
s = if T then F else T
t = if s then T else T
u = if F then T else T
```

entonces podemos justificar que:

$$\texttt{if t then F else F} \rightarrow \texttt{if u then F else F}$$

con el siguiente árbol de derivación:

$$\dfrac{\dfrac{\dfrac{}{s \rightarrow \texttt{F}}\,(\text{E-IfTrue})}{t \rightarrow u}\,(\text{E-If})}{\texttt{if } t \texttt{ then F else F} \rightarrow \texttt{if } u \texttt{ then F else F}}\,(\text{E-If})$$

3.3. Inducción sobre una derivación.

3.3.1. Idea general.

Cuando queremos demostrar propiedades de la relación de evaluación ($\rightarrow$), lo hacemos por **inducción sobre la derivación.**

El objetivo es: demostrar que un predicado P se cumple para cualquier derivación D.

La hipótesis inductiva (HI) será: si $P(C)$ vale para todas las derivaciones inmediatas C de D, entonces $P(D)$ también vale.

3.3.2. Ejemplo. Teorema: Determinismo de la evaluación de paso chico.

Si $t \rightarrow t'$ y $t \rightarrow t''$, entonces $t' = t''$.

Pasos a seguir.

1. Elegimos sobre qué derivación vamos a hacer inducción (por ejemplo, $t \rightarrow t'$).
2. Hacemos análisis por casos sobre la última regla aplicada en la derivación. Pudiendo usar la HI en los casos que contienen subderivaciones.

Demostración.

Haremos inducción sobre la derivación $t \rightarrow t'$.

- Si la última regla aplicada es `E-IfTrue`, entonces la forma de t es `if T then` t_2 `else` t_3 y luego $t' = t_2$.

  En la derivación $t \rightarrow t''$ la última regla aplicada no puede ser `E-IfFalse` por la forma que tiene t: `if T then` t_2 `else` t_3.

  Tampoco puede ser `E-If`, ya que esa regla requiriría que $T \rightarrow t'_1$, lo cual no puede ser porque T es un valor.

  Por lo tanto, la única regla aplicada es `E-IfTrue`. Así, por esta regla concluimos que $t'' = t_2$, con lo cual $t' = t''$.

- Si la última regla aplicada es `E-IfFalse`, el razonamiento es similar al caso anterior.

- Si la última regla aplicada es `E-If`, entonces t tiene la forma `if` t_1 `then` t_2 `else` t_3, donde $t_1 \rightarrow t'_1$ y por lo tanto $t_2 =$ `if` t'_1 `then` t_2 `else` t_3.

  En la derivación $t \rightarrow t''$, la última regla no pudo ser `E-IfTrue`, dado que $t_1 \rightarrow t'_1$, es decir, no puede ser T. Tampoco pudo aplicarse `E-IfFalse`.

  La única regla aplicada es `E-If`, es decir que $t_1 \rightarrow t''_1$ y $t'' =$ `if` t''_1 `then` t_2 `else` t_3.

  Por HI, dado que $t_1 \rightarrow t'_1$ es una subderivación de $t \rightarrow t'$ y $t_1 \rightarrow t''_1$, entonces $t'_1 = t''_1$ y luego $t' = t''$.

3.4. Más definiciones y resultados.

3.4.1. Definición. Forma normal.

Un término t está en **forma normal** si no se le puede aplicar ninguna regla de evaluación. O sea, t está en forma normal si no existe t' tal que $t \rightarrow t'$.

Para nuestro lenguaje simple, las formas normales son `T` y `F` (los valores).

Teorema. Todo valor está en forma normal. En general el converso no vale (por ejemplo, errores de ejecución), pero para nuestro lenguaje vale que si t está en forma normal, entonces t es un valor. La prueba se puede hacer por inducción estructural sobre t en el contrarrecíproco.

3.4.2. Evaluación de pasos múltiples.

La relación de **pasos múltiples** $\to^*$ es la clausura reflexivo-transitiva de $\to$. Es decir, es la menor relación tal que:

$$\frac{t \to t'}{t \to^* t'} \qquad \frac{}{t \to^* t} \qquad \frac{t \to^* t' \qquad t' \to^* t''}{t \to^* t''}$$

3.4.3. Teorema. Unicidad de formas normales.

Si $t \to^* u$ y $t \to^* u'$, donde u y u' son formas normales, entonces $u = u'$.

3.4.4. Teorema. La evaluación termina.

Para todo término t hay una forma normal t' tal que $t \to^* t'$.

3.4.5. Teorema. Determinismo en big-step.

Si $t \Downarrow v$ y $t \Downarrow v'$ entonces $v = v'$.

3.4.6. Teorema. Terminación en big-step.

Para todo término t, existe v tal que $t \Downarrow v$.

3.4.7. Teorema. Equivalencia de paso grande y chico.

Para todo término t y valor v, $t \Downarrow v$ si y sólo si $t \to^* v$.

3.5. Semántica del lenguaje de expresiones aritméticas.

3.5.1. Definición del lenguaje.

Trabajamos ahora con el lenguaje de expresiones aritméticas completo:

```
t -> T | F | if t then t else t | 0 | succ t | pred t | iszero t
```

Para definir los valores agregamos una nueva categoría sintáctica de valores numéricos:

```
v -> T | F | nv
nv -> 0 | succ nv
```

3.5.2. Nuevas reglas de evaluación small-step.

Vamos a definir la relación de evaluación para el lenguaje completo, agregando reglas a las existentes:

$$\frac{t_1 \to t_1'}{\texttt{succ}\ t_1 \to \texttt{succ}\ t_1'} \quad \text{(E-Succ)}$$

$$\texttt{pred}\ 0 \to 0 \quad \text{(E-PredZero)}$$

$$\texttt{pred}\ (\texttt{succ}\ nv_1) \to nv_1 \quad \text{(E-PredSucc)}$$

$$\frac{t_1 \to t_1'}{\texttt{pred}\ t_1 \to \texttt{pred}\ t_1'} \quad \text{(E-Pred)}$$

$$\texttt{iszero}\ 0 \to \texttt{T} \quad \text{(E-isZeroZero)}$$

$$\texttt{iszero}\ (\texttt{succ}\ nv_1) \to \texttt{F} \quad \text{(E-IsZeroSucc)}$$

$$\frac{t_1 \to t_1'}{\texttt{iszero}\ t_1 \to \texttt{iszero}\ t_1'} \quad \text{(E-IsZero)}$$

Observaciones.

- Notar el rol que juega la categoría sintáctica nv en la estrategia de evaluación. Por ejemplo, no se puede usar `E-PredSucc` para concluir que `pred (succ (pred 0)) ->pred 0`
- Notar que términos como `succ F` son formas normales, pero **no son valores.**
- Si t es una forma normal pero no es un valor, decimos que t está **atascado (stuck).** Un término atascado se puede pensar como error de run-time. No se puede seguir la ejecución porque se llegó a un estado sin sentido.

(hacer ejercicios a lo largo del apunte).

4. Unidad 3.2 - Semántica Operacional.

4.1. Semántica operacional de lenguaje imperativo simple.

Veremos la semántica operacional de un lenguaje imperativo simple, donde modelaremos los efectos laterales de:

- Estado
- Errores
- Entrada/Salida

4.1.1. Sintaxis del lenguaje.

Definimos la sintaxis del lenguaje imperativo con las siguientes reglas:

$$\texttt{ie} \to \texttt{nv} \mid \texttt{var} \mid \texttt{-ie} \mid \texttt{ie + ie} \mid \texttt{ie} -_b \texttt{ie} \mid \texttt{ie x ie} \mid \texttt{ie / ie}$$
$$\texttt{be} \to \texttt{true} \mid \texttt{false} \mid \texttt{ie = ie} \mid \texttt{ie < ie} \mid \texttt{ie > ie} \mid \texttt{be} \wedge \texttt{be} \mid \texttt{be} \vee \texttt{be} \mid \neg\texttt{be}$$
$$\texttt{comm} \to \texttt{skip} \mid \texttt{var := ie} \mid \texttt{comm ; comm} \mid \texttt{if be then comm else comm} \mid \texttt{while be comm}$$

Observaciones.

- Notación. ie $\to$ integral expression, be $\to$ boolean expression, comm $\to$ comando.
- El operador `skip` es no hacer nada.
- El operador `:=` es de asignación de variables.
- El operador ; es de secuenciación, es decir, ejecutar un comando y luego otro.

4.1.2. Estado.

El **estado** representa la memoria del programa: qué valor tiene cada variable. El lenguaje solo contiene variables enteras. La noción de estado se puede modelar con la siguiente definición:

$$\Sigma = Var \to nv$$

Es decir, un estado es una función total entre identificadores y enteros: cada variable tiene asignado un numero entero.

Para modificar el estado utilizaremos la siguiente notación:

$$[\sigma | v : n]$$

que representa 'igual que el estado σ, pero la variable v vale n'.

4.1.3. Relaciones de evaluación de paso grande.

Queremos formalizar como se evalúan las expresiones/comandos en un estado.

1. **Expresiones enteras.**
   - Tipo de la relación:
     $$\Downarrow_i \subseteq (ie \times \Sigma) \times nv$$
     se lee: 'una expresión entera y un estado producen un número'.
   - Regla para variables:
     $$\frac{}{(v, \sigma) \Downarrow_i \sigma(v)} \quad \text{(Var)}$$

- Regla para la suma:

$$\frac{(e_0, \sigma) \Downarrow_i n_0 \qquad (e_1, \sigma) \Downarrow_i n_1}{(e_0 + e_1, \sigma) \Downarrow_i n_0 + n_1} \qquad \text{(Add)}$$

- Regla para la resta:

$$\frac{(e_0, \sigma) \Downarrow_i n_0 \qquad (e_1, \sigma) \Downarrow_i n_1}{(e_0 -_b e_1, \sigma) \Downarrow_i n_0 - n_1} \qquad \text{(Sub)}$$

- Regla para el producto:

$$\frac{(e_0, \sigma) \Downarrow_i n_0 \qquad (e_1, \sigma) \Downarrow_i n_1}{(e_0 \times e_1, \sigma) \Downarrow_i n_0 \times n_1} \qquad \text{(Prod)}$$

- Regla para la división producto:

$$\frac{(e_0, \sigma) \Downarrow_i n_0 \qquad (e_1, \sigma) \Downarrow_i n_1}{(e_0 / e_1, \sigma) \Downarrow_i n_0 / n_1} \qquad \text{(Div)}$$

2. **Expresiones booleanas.**

- Tipo de la relación:

$$\Downarrow_b \in (be \times \Sigma) \times bv$$

donde bv es un *valor booleano*, es decir, $bv = \{true, false\}$ y la relación se lee como 'una expresión booleana y un estado producen un valor booleano'.

- Regla para la igualdad:

$$\frac{(e_0, \sigma) \Downarrow_i n_0 \qquad (e_1, \sigma) \Downarrow_i n_1}{(e_0 = e_1, \sigma) \Downarrow_b n_0 = n_1} \qquad \text{(Eq)}$$

- Regla para relación menor:

$$\frac{(e_0, \sigma) \Downarrow_i n_0 \qquad (e_1, \sigma) \Downarrow_i n_1}{(e_0 < e_1, \sigma) \Downarrow_b n_0 < n_1} \qquad \text{(Low)}$$

- Regla para relación mayor:

$$\frac{(e_0, \sigma) \Downarrow_i n_0 \qquad (e_1, \sigma) \Downarrow_i n_1}{(e_0 > e_1, \sigma) \Downarrow_b n_0 > n_1} \qquad \text{(Gre)}$$

- Regla para conjunción:

$$\frac{(e_0, \sigma) \Downarrow_b b_0 \qquad (e_1, \sigma) \Downarrow_b b_1}{(e_0 \wedge e_1, \sigma) \Downarrow_b b_0 \wedge b_1} \qquad \text{(And)}$$

- Regla para disyunción:

$$\frac{(e_0, \sigma) \Downarrow_b b_0 \qquad (e_1, \sigma) \Downarrow_b b_1}{(e_0 \vee e_1, \sigma) \Downarrow_b b_0 \vee b_1} \qquad \text{(Or)}$$

- Regla para negación:

$$\frac{(e_0, \sigma) \Downarrow_b b_0}{(\neg e_0 \vee, \sigma) \Downarrow_b \neg b_0} \qquad \text{(Neg)}$$

3. **Evaluación para comandos.** Cuando hablamos de comandos (asignaciones, secuencias, etc) queremos darles un significado formal en términos de cómo transforman un estado σ.

   - Tipo de la relación:
   $$\Rightarrow \in (\texttt{comm}, \Sigma) \times \Sigma$$
   y se lee como 'el comando c, ejecutado en el estado σ, produce el estado σ''.

   - Regla para la asignación:
   $$\frac{(e, \sigma) \Downarrow_i n}{(x := e, \sigma) \Rightarrow [\sigma | x : n]} \quad \text{(Ass)}$$

   - Regla para el skip:
   $$\frac{}{(\texttt{skip}, \sigma) \Rightarrow \sigma} \quad \text{(Skip)}$$

   - Regla para el secuenciamiento:
   $$\frac{(c_0, \sigma) \Rightarrow \sigma' \qquad (c_1, \sigma') \Rightarrow \sigma''}{(c_0; c_1, \sigma) \Rightarrow \sigma''} \quad \text{(Seq)}$$

   - Regla para el if con condición verdadera:
   $$\frac{(e, \sigma) \Downarrow_b \texttt{true} \qquad (c_1, \sigma) \Rightarrow \sigma'}{(\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, \sigma) \Rightarrow \sigma'} \quad \text{(If-T)}$$

   - Regla para el if con condición falsa:
   $$\frac{(e, \sigma) \Downarrow_b \texttt{false} \qquad (c_2, \sigma) \Rightarrow \sigma'}{(\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, \sigma) \Rightarrow \sigma'} \quad \text{(If-F)}$$

   - Regla para el while con condición verdadera:
   $$\frac{(e, \sigma) \Downarrow_b \texttt{true} \qquad (c, \sigma) \Rightarrow \sigma' \qquad (\texttt{while } e\ c, \sigma') \Rightarrow \sigma''}{(\texttt{while } e\ c, \sigma) \Rightarrow \sigma''} \quad \text{(While-T)}$$

   - Regla para el while con condición falsa:
   $$\frac{(e, \sigma) \Downarrow_b \texttt{false}}{(\texttt{while } e\ c, \sigma) \Rightarrow \sigma} \quad \text{(While-F)}$$

4.1.4. Evaluación de paso chico para comandos (small-step).

Vamos a definir las reglas en **paso chico** ya que nos ayudarán a:

- Describir la ejecución paso a paso, como una máquina abstracta.
- Relacionar un programa con su siguiente configuración.
- Mostrar toda la 'traza' de ejecución: para observar la dinámica interna del programa y poder modelar la no terminación, concurrencia y errores en tiempo de ejecución.

Tipo de la relación.

$$\rightsquigarrow \in (\texttt{comm} \times \Sigma) \times (\texttt{comm} \times \Sigma)$$

y se lee como 'el par (comando, estado) hace un paso de ejecución hacia (nuevo comando, nuevo estado)'.

Observaciones.

- Si la ejecución termina lo hace en una configuración de la forma ($\texttt{skip}$, σ), para algún σ.
- Las ejecuciones de la asignación y `skip` terminan en un paso, por eso son similares a las de paso grande.
- Regla para el secuenciamiento:

$$\frac{(c_0, \sigma) \rightsquigarrow (c_0', \sigma')}{(c_0; c_1, \sigma) \rightsquigarrow (c_0'; c_1, \sigma')} \quad \text{(Seq1)}$$

- Regla para el secuenciamiento con primer comando skip:

$$\frac{}{(\texttt{skip}; c_1, \sigma) \rightsquigarrow (c_1, \sigma)} \quad \text{(Seq2)}$$

- Condicionales (mantenemos evaluación de paso grande para expresiones):

$$\frac{e \Downarrow_b \texttt{true}}{(\texttt{if } e \texttt{ then } c_0 \texttt{ else } c_1, \sigma) \rightsquigarrow (c_0, \sigma)} \quad \text{(If-T)}$$

$$\frac{e \Downarrow_b \texttt{false}}{(\texttt{if } e \texttt{ then } c_0 \texttt{ else } c_1, \sigma) \rightsquigarrow (c_1, \sigma)} \quad \text{(If-F)}$$

- Regla para bucles:

$$\frac{e \Downarrow_b \texttt{false}}{(\texttt{while } e\ c, \sigma) \rightsquigarrow (\texttt{skip}, \sigma)} \quad \text{(While-F)}$$

$$\frac{e \Downarrow_b \texttt{true}}{(\texttt{while } e\ c, \sigma) \rightsquigarrow (c; \texttt{while } e\ c, \sigma)} \quad \text{(While-T)}$$

4.1.5. Extensión de la semántica: manejo de errores.

En la semántica operativa de nuestro lenguaje, necesitamos extender las reglas para modelar **errores en tiempo de ejecución,** como división por cero o comparaciones inválidas.

1. **Errores en expresiones enteras.** Agregamos un valor especial que representa un error entero, las expresiones enteras que fallen devolverán este valor:

   $$err_i$$

   El tipo de la relación de evaluación entera antes tenía la siguiente forma:

   $$\Downarrow_i \in (ie \times \Sigma) \times nv$$

   Ahora, introduciendo los errores, tiene el siguiente tipo:

   $$\Downarrow_i \in (ie \times \Sigma) \times (nv \cup \{err_i\})$$

   es decir, una expresión entera puede evaluar a un número o a un error.

   - Regla de evaluación para error de división por cero:

   $$\frac{(e_0, \sigma) \Downarrow_i n_0 \qquad (e_1, \sigma) \Downarrow_i 0}{(e_0/e_1, \sigma) \Downarrow_i err_i} \quad \text{(Div-0)}$$

   - Regla de evaluación para divisiones que no generan error:

   $$\frac{(e_0, \sigma) \Downarrow_i n_0 \qquad (e_1, \sigma) \Downarrow_i n_1 \qquad n_1 \neq 0}{(e_0/e_1, \sigma) \Downarrow_i n_0/n_1} \quad \text{(Div-1)}$$

- Reglas para propagar el error en la división:

$$\frac{(e_0, \sigma) \Downarrow_i err_i}{(e_0/e_1, \sigma) \Downarrow_i err_i} \quad \text{(Div-2)}$$

$$\frac{(e_0, \sigma) \Downarrow_i n_0 \qquad (e_1, \sigma) \Downarrow_i err_i}{(e_0/e_1, \sigma) \Downarrow_i err_i} \quad \text{(Div-3)}$$

2. **Errores en expresiones booleanas.** Agregamos un valor especial que representa un error booleano, las expresiones booleanas que fallen devolverán este valor:

$$err_b$$

El tipo de la relación de evaluación booleana antes tenía la siguiente forma:

$$\Downarrow_b \in (be \times \Sigma) \times bv$$

Ahora, introduciendo los errores, tiene el siguiente tipo:

$$\Downarrow_b \in (be \times \Sigma) \times (bv \cup \{err_b\})$$

es decir, una expresión booleana puede evaluar a un valor booleano o a un error.

- Regla para error en la igualdad:

$$\frac{(e_0, \sigma) \Downarrow_i err_i}{(e_0 = e_1, \sigma) \Downarrow_b err_b} \quad \text{(Eq-0)}$$

- Regla para error en la conjunción:

$$\frac{(e_0, \sigma) \Downarrow_b err_b}{(e_0 \wedge e_1, \sigma) \Downarrow_b err_b} \quad \text{(And-0)}$$

3. **Errores en comandos.** Agregamos un valor especial que representa un error en comandos, los comandos que fallen devolverán este valor:

$$err_c$$

El tipo de la relación de paso chico en comandos antes tenía la siguiente forma:

$$\rightsquigarrow \in (comm \times \Sigma) \times (comm \times \Sigma)$$

Ahora, introduciendo los errores, tiene el siguiente tipo:

$$\rightsquigarrow \in (comm \times \Sigma) \times ((comm \cup \{err_c\}) \times \Sigma)$$

es decir, un comando al dar un paso puede transformarse en: otro comando con un nuevo estado; un error err_c representando fallo en la ejecución.

- Regla para errores en la asignación:

$$\frac{(e, \sigma) \Downarrow_i err_i}{(x := e, \sigma) \rightsquigarrow (err_c, \sigma)} \quad \text{(Ass-0)}$$

- Regla para errores en el secuenciamiento:

$$\frac{(c_0, \sigma) \rightsquigarrow (err_c, \sigma')}{(c_0; c_1, \sigma) \rightsquigarrow (err_c, \sigma')} \quad \text{(Seq-0)}$$

4.1.6. Extensión de la semántica: entrada y salida (E/S).

Para modelar entradas que interactúan con el mundo externo (leyendo entradas o mostrando salidas), se extiende la semántica operacional agregando **etiquetas** a las transiciones:

- La etiqueta $n?$ indica la **entrada** de un entero n.
- La etiqueta $n!$ indica la **salida** (impresión) de un entero n.
- La etiqueta τ representa una transición **silenciosa** (sin interacción en el entorno). Escribiremos $x \rightsquigarrow y$ en lugar de $x \rightsquigarrow^{\tau} y$.

Hasta ahora, las transiciones de comandos de paso chico eran del siguiente tipo:

$$\rightsquigarrow \in (comm \times \Sigma) \times ((comm \cup \{err_c\}) \times \Sigma)$$

Pero introduciendo E/S, las transiciones se enriquecen con una etiqueta l y quedan con el siguiente tipo:

$$\rightsquigarrow \in (comm \times \Sigma) \times l \times ((comm \cup \{err_c\}) \times \Sigma)$$

Además, agregamos dos nuevos comandos al lenguaje:

```
comm := ... | input var | print ie
```

donde `input var` lee un entero desde la entrada y lo guarda en la variable *var*, y `print ie` evalúa la expresión entera *ie* y la envía a la salida.

Además, agregamos reglas semánticas de paso chico para estos comandos:

- Regla para la lectura de un entero:

$$\frac{}{(\texttt{input}\ v, \sigma) \rightsquigarrow^{n?} (\texttt{skip}, [\sigma | v : n])} \quad \text{(Input)}$$

- Regla para impresión de un entero:

$$\frac{(e, \sigma) \Downarrow_i n}{(\texttt{print}\ e, \sigma) \rightsquigarrow^{n!} (\texttt{skip}, \sigma)} \quad \text{(Print)}$$

5. Unidad 4 - Lambda Cálculo.

5.1. Introducción.

5.1.1. Línea del tiempo.

5.1.2. Definición.

El **Cálculo Lambda** (λ-cálculo) es un sistema formal matemático desarrollado en la década de 1930 por Alonzo Church. Se puede ver como un lenguaje de programación minimalista centrado en la abstracción de funciones y su aplicación.

El λ-cálculo está diseñado para estudiar la **computabilidad,** demostrando ser **Turing-completo,** lo que significa que es capaz de expresar cualquier función computable, al igual que una Máquina de Turing. Los elementos básicos del λ-cálculo son:

1. **Variables:** nombres o marcadores de posición (por ejemplo, x, y, z).
2. **Abstracciones lambda:** la forma de definir una función anónima (sin nombre).

   La sintaxis $\lambda x.M$, donde λ introduce la función, x es el parámetro (variable ligada) y M es el cuerpo o expresión que define lo que hace la función.

   Por ejemplo, la función $\lambda x.x$ representa la función identidad ($f(x) = x$).
3. **Aplicaciones:** la acción de llamar o ejecutar una función con un argumento. La sintaxis es MN, donde M es la función y N el argumento.

   Por ejemplo, $(\lambda x.x)a$ significa aplicar la función identidad $\lambda x.x$ al argumento a.

5.1.3. Sintaxis del lambda cálculo.

Se denota con $\bigwedge$ al conjunto de λ-términos, definido inductivamente por las siguientes reglas:

$$\frac{x \in X}{x \in \bigwedge} \qquad \frac{t \in \bigwedge \quad u \in \bigwedge}{(t\ u) \in \bigwedge} \qquad \frac{x \in X \quad t \in \bigwedge}{(\lambda x.t) \in \bigwedge}$$

donde X es un conjunto infinito de identificadores/variables.

Ejemplos: $x, (xy), (\lambda x.x), (\lambda x.(\lambda y.x)), (\lambda x.(\lambda y.((zx)y)))$

Convenciones.

- Se utilizarán letras mayúsculas para denotar λ-términos.
- La aplicación asocia a izquierda. Escribimos $M\ N\ T$ en lugar de $((MN)T)$.
- La abstracción se extiende tanto como sea posible. Escribimos $\lambda x.M\ N$ en lugar de $(\lambda x.M\ N)$.
- Se pueden juntar varios λ en uno solo: Escribimos $\lambda\ x_1\ x_2 ... x_n\ .\ M$ en lugar de $(\lambda\ x_1 (\lambda\ x_2 (... \lambda x_n.M) ...)$

5.2. Variables libres, ligadas, equivalencia y sustitución.

5.2.1. Variables libres y ligadas.

En el λ-cálculo, la distinción entre variables **libres** y **ligadas** es fundamental para entender el alcance de los parámetros y el proceso de β-reducción.

- Una variable está **ligada** si es el parámetro de una λ-abstracción, o si está dentro del cuerpo de una λ-abstracción que la ha declarado.
- Una variable está **libre** si aparece en un término sin haber sido declarada como parámetro en ninguna λ-abstracción que la contenga.

Formalmente se definen el conjunto de variables libres (FV) y ligadas (BV) de una expresión (e) recursivamente sobre la estructura del término λ:

- $FV(e)$ es el conjunto de ocurrencias de **variables libres** en e, definido como:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.M) &= FV(M) - \{x\} \\ FV(M\ N) &= FV(M) \cup FV(N) \end{aligned}$$

- $BV(e)$ es el conjunto de ocurrencias de **variables ligadas** en e, definido como:

$$\begin{aligned} BV(x) &= \emptyset \\ BV(\lambda x.M) &= BV(M) \cup \{x\} \\ BV(M\ N) &= BV(M) \cup BV(N) \end{aligned}$$

Más definiciones y convenciones.

- Llamamos **ocurrencia de ligadura** a la variable x que aparece inmediatamente después del símbolo λ, es decir, en la posición $\lambda x.M$. Es la variable que actúa como parámetro formal de la función.
- Un **término cerrado** es un término que no contiene variables libres, es decir, tal que $FV(e) = \emptyset$. Solo depende de sus propias λ-abstracciones.

Ejemplo. Encontrar las ocurrencias de variables libres, ligadas y de ligadura del término:

$$(\lambda y.y\ x\ (\lambda x.y\ (\lambda y.z)\ x))\ v\ w$$

1. **Variables de ligadura:** son las que están justo después de λ, es decir, y (en la primera abstracción, x (en la segunda), y (en la tercera).
2. **Abstracciones presentes:**
   - $A_1 = (\lambda y.M_1)$, donde $M_1 = yx(\lambda x.y(\lambda y.z)x)$
   - $A_2 = (\lambda x.M_2)$, donde $M_2 = y(\lambda y.z)x$
   - $A_3 = (\lambda y.z)$
3. **Variables ligadas $BV(e)$:** y (ligada por A_1), x (ligada por A_2), y (ligada por A_3). Por lo tanto:
$$BV(e) = \{x, y\}$$
4. **Variables libres $FV(e)$:** aquellas que no son parámetros de ninguna λ-abstracción:

Variable	Ocurrencia	Ligada por	¿Libre?
v	Aplicación externa	Ninguna	Si
w	Aplicación externa	Ninguna	Si
x	En M_1	Ninguna (fuera del alcance de A_2)	Si
z	En A_3	Ninguna (no es parámetro)	Si
y	En M_1 y M_2	A_1 (la más externa)	No
x	En A_2	A_2	No
y	En A_3	A_3	No

Notar que la variable x que está etnre y y A_2 es libre porque se encuentra fuera del cuerpo de la abstracción A_2 que liga x y no está ligada por A_1. Por lo tanto,

$$FV(e) = \{x, z, v, w\}$$

5.2.2. Equivalencia sintáctica.

La **equivalencia sintáctica** $\equiv$ es la forma más estricta de igualdad entre términos λ. Se denota como $M \equiv N$ si y sólo si M **es exactamente el mismo término que** N en su estructura de símbolos.

En términos sencillos, dos expresiones λ son sintácticamente equivalentes si, y sólo si, son idénticas carácter por carácter.

Observación. Aunque esta definición es clara, en el λ-cálculo pocas veces es útil para el propósito de la computación, porque no captura la idea de que dos funciones pueden ser esencialmente la misma, aunque usen diferentes nombres para sus parámetros.

Por ejemplo, los términos $\lambda x.x$ y $\lambda y.y$ son sintácticamente distintos ($\lambda x.x \not\equiv \lambda y.y$) aunque ambos representan la función identidad.

Para la computación, deseamos que se consideren equivalentes, lo cual se logra mediante la α-equivalencia, que se verá luego.

5.2.3. Sustitución.

Sean M y N términos, se define $M[N/x]$ como el resultado de **reemplazar toda ocurrencia libre de la variable** x **en el término** M **por el término** N**.**

La regla de sustitución se define por inducción sobre la estructura del término M, y debe ser segura, es decir, evitar la captura de variables ligadas (que una variable libre de N se capture accidentalmente en M).

Definimos $M[N/x]$ por inducción sobre M como:

$$
\begin{aligned}
x[N/x] &\equiv N \\
y[N/x] &\equiv y \\
(P\ Q)[N/x] &\equiv (P[N/x]\ Q[N/x]) \\
(\lambda x.P)[N/x] &\equiv (\lambda x.P) \\
(\lambda y.P)[N/x] &\equiv (\lambda y.P) && x \notin FV(P) \\
(\lambda y.P)[N/x] &\equiv (\lambda y.P[N/x]) && x \in FV(P) \wedge y \notin FV(N) \\
(\lambda y.P)[N/x] &\equiv \lambda z.(P[z/y])[N/x] && x \in FV(P) \wedge y \in FV(N)
\end{aligned}
$$

donde $z \notin FV(N\ P)$ y $x \not\equiv y$.

Ejemplos.

- $(\lambda y.x\ (\lambda w.v\ w\ x))[(u\ v)/x] \rightarrow$ llamaremos $P = x\ (\lambda w.v\ w\ x)$.

$$
\begin{aligned}
(\lambda y.P)[(u\ v)/x] &= (\lambda y.P[(u\ v)/x]) && \text{6to caso} \\
&= (\lambda y.(x\ (\lambda w.v\ w\ x)[(u\ v)/x])) \\
&= (\lambda y.x[(u\ v)/x]\ (\lambda w.v\ w\ x)[(u\ v)/x]) && \text{3er caso} \\
&= (\lambda y.(u\ v)(\lambda w.v\ w\ x)[(u\ v)/x] && \text{1er caso} \\
&= (\lambda y.(u\ v)(\lambda w.(v\ w\ x)[(u\ v)/x]) && \text{6to caso} \\
&= (\lambda y.(u\ v)(\lambda w.v\ w\ (u\ v))) && \text{1er y 2do caso}
\end{aligned}
$$

5.2.4. α-equivalencia.

La **α-equivalencia** es la primera de las tres reglas fundamentales de equivalencia en el λ-cálculo (junto con la β y la η). Esta relación captura la idea de que la identidad de una función no depende del nombre que le demos a su parámetro.

Los términos P y Q son α-equivalentes, y se denota $P \equiv_\alpha Q$, si solo difieren en el **nombre de sus variables ligadas** (parámetros de las λ-abstracciones).

Regla de α-conversión.

La α-conversión es la operación que permite renombrar una variable ligada de forma segura:

1. **Término inicial:** sea un término P que contiene la subexpresión $\lambda x.M$.
2. **Operación:** se reemplaza $\lambda x.M$ por $\lambda y.(M[y/x])$.
3. **Condición de seguridad:** la nueva variable y debe ser *fresca*; es decir, y no debe ocurrir libremente en el cuerpo original M ($y \notin FV(M)$). Esta condición garantiza que la nueva variable y solo ligue a las ocurrencias de x que se pretendían, evitando la captura accidental de variables.

Si un término P puede transformarse en un termino Q mediante una serie finita de estas α-conversiones, decimos que $P \equiv_\alpha Q$.

Ejemplo.

$$\begin{aligned} \lambda x\ y.x\ (x\ y) &\equiv \lambda x.(\lambda y.x\ (x\ y)) \\ &\equiv_\alpha \lambda x.(\lambda v.x\ (x\ v)) && \text{conversión en la abstracción interna} \\ &\equiv_\alpha \lambda u.(\lambda v.u\ (u\ v)) && \text{conversión en la abstracción externa} \\ &\equiv \lambda u\ v.u\ (u\ v) \end{aligned}$$

Propiedades de $\equiv_\alpha$.

1. Si $P \equiv_\alpha Q$, entonces $FV(P) = FV(Q)$.
2. La relación $\equiv_\alpha$ es una relación de equivalencia, es decir, para cualesquiera P, Q, R:
   - $P \equiv_\alpha P$
   - $P \equiv_\alpha Q \Rightarrow Q \equiv_\alpha P$
   - $P \equiv_\alpha Q \wedge Q \equiv_\alpha R \ \Rightarrow\ P \equiv_\alpha R$
3. La relación $\equiv_\alpha$ es preservada por la sustitución:

$$P \equiv_\alpha P' \wedge Q \equiv_\alpha Q' \ \Rightarrow\ P[Q/x] \equiv_\alpha P'[Q'/x]$$

5.3. Semántica en el λ-cálculo.

La semántica del λ-cálculo, es decir, cómo se evalúan los términos, se define por las reglas de reducción. Un paso de evaluación es la **aplicación de una abstracción** (función) a un argumento. La regla fundamental de evaluación es la β-reducción.

5.3.1. β-reducción.

(página 15/32)