Análisis de Lenguajes de Programación

Ignacio Rimini

August 2025

Índice

1. Unidad 1 - Sintaxis. **5**
1.1. Definición. Sintaxis. . . . 5
1.2. Ejemplos de sintaxis. . . . 5
1.2.1. Ejemplo de sintaxis abstracta. . . . 5
1.2.2. Ejemplo de sintaxis concreta. . . . 6
1.3. Gramáticas Libres de Contexto (CFG). . . . 6
1.3.1. Introducción. . . . 6
1.3.2. Definición. Grámatica Libre de Contexto. . . . 6
1.3.3. Ejemplo de gramática libre de contexto. . . . 6
1.3.4. Definición. Relación de derivación ($\Rightarrow$). . . . 7
1.3.5. Definición. Relación de derivación reflexiva-transitiva ($\Rightarrow^*$). . . . 7
1.3.6. Definición. Lenguaje generado por una gramática. . . . 7
1.3.7. Ejemplo de derivación a partir de gramática. . . . 7
1.3.8. Propiedades de lenguajes libres de contexto. . . . 8
1.3.9. Definición. Árbol de parseo. . . . 8
1.4. Gramáticas ambiguas. . . . 9
1.4.1. Definición. Gramática ambigua. . . . 9
1.4.2. Desambiguar gramáticas. . . . 9
1.5. Árboles de sintaxis abstracta. . . . 11
1.5.1. Introducción. . . . 11
1.5.2. Implementación de AST. . . . 11
1.5.3. Lenguaje de booleanos y naturales. . . . 11
1.5.4. Conjunto de términos. . . . 11

2. Unidad 2 - Analizadores Sintácticos (Parsers). **13**
2.1. Introducción. . . . 13
2.1.1. ¿Qué es un parser? . . . 13
2.1.2. El tipo de los parsers en Haskell. . . . 13
2.2. Parsers varios y operaciones. . . . 14
2.2.1. Parsers básicos. . . . 14
2.2.2. Combinadores de parsers: CHOICE. . . . 15
2.2.3. Combinadores de parsers: SECUENCIAMIENTO. . . . 15
2.2.4. Primitivas derivadas. . . . 16
2.2.5. Parser ignorando espacios. . . . 18
2.3. Parser de gramáticas. . . . 20
2.3.1. De gramática a parser en Haskell. . . . 20
2.3.2. Asociatividad de operadores y recursión a izquierda. . . . 21

3. Unidad 3.1 - Semántica. **22**
3.1. Introducción. . . . 22
3.1.1. Definición formal de un lenguaje. . . . 22
3.1.2. Beneficios de la semántica formal. . . . 22

3.1.3. Enfoques clásicos de semántica. . . . 22
3.2. Tipos de semántica operacional. . . . 22
3.2.1. Ejemplo de semántica operacional big-step. . . . 23
3.2.2. Ejemplo de árbol de derivación para semántica big-step. . . . 24
3.2.3. Relación de evaluación small-step. . . . 24
3.2.4. Ejemplo de árbol de derivación para semántica small-step. . . . 24
3.3. Inducción sobre una derivación. . . . 25
3.3.1. Idea general. . . . 25
3.3.2. Ejemplo. Teorema: Determinismo de la evaluación de paso chico. . . . 25
3.4. Más definiciones y resultados. . . . 25
3.4.1. Definición. Forma normal. . . . 25
3.4.2. Evaluación de pasos múltiples. . . . 26
3.4.3. Teorema. Unicidad de formas normales. . . . 26
3.4.4. Teorema. La evaluación termina. . . . 26
3.4.5. Teorema. Determinismo en big-step. . . . 26
3.4.6. Teorema. Terminación en big-step. . . . 26
3.4.7. Teorema. Equivalencia de paso grande y chico. . . . 26
3.5. Semántica del lenguaje de expresiones aritméticas. . . . 26
3.5.1. Definición del lenguaje. . . . 26
3.5.2. Nuevas reglas de evaluación small-step. . . . 26

4. Unidad 3.2 - Semántica Operacional. 28
4.1. Semántica operacional de lenguaje imperativo simple. . . . 28
4.1.1. Sintaxis del lenguaje. . . . 28
4.1.2. Estado. . . . 28
4.1.3. Relaciones de evaluación de paso grande. . . . 28
4.1.4. Evaluación de paso chico para comandos (small-step). . . . 30
4.1.5. Extensión de la semántica: manejo de errores. . . . 31
4.1.6. Extensión de la semántica: entrada y salida (E/S). . . . 33

5. Unidad 4.1 - Lambda Cálculo. 34
5.1. Introducción. . . . 34
5.1.1. Línea del tiempo. . . . 34
5.1.2. Definición. . . . 34
5.1.3. Sintaxis del lambda cálculo. . . . 34
5.2. Variables libres, ligadas, equivalencia y sustitución. . . . 34
5.2.1. Variables libres y ligadas. . . . 34
5.2.2. Equivalencia sintáctica. . . . 36
5.2.3. Sustitución. . . . 36
5.2.4. α-equivalencia. . . . 36
5.3. Semántica en el λ-cálculo. . . . 37
5.3.1. β-reducción. . . . 37
5.3.2. Estrategias de reducción. . . . 38
5.3.3. Semántica operacional del λ-cálculo. . . . 38
5.3.4. Definición. Forma normal β. . . . 38
5.3.5. Propiedades de la β-reducción ($\rightarrow_\beta^*$). . . . 39
5.4. Consistencia y equivalencia en el λ-cálculo. . . . 39
5.4.1. Confluencia y teorema de Church-Rosser. . . . 39
5.4.2. Definición. β-equivalencia. . . . 39
5.4.3. Propiedades de la β-equivalencia. . . . 40
5.4.4. Extensionalidad. . . . 40
5.4.5. Definición. η-redex, η-contracción y relación $=_{\beta\eta}$. . . . 40
5.5. Normalización. . . . 41
5.5.1. Definición. Términos normalizantes. . . . 41
5.5.2. Reducción normal. . . . 42
5.5.3. Formas neutrales y normales. . . . 42

5.5.4. Reglas de evaluación para la reducción normal. 42
5.6. Resumen. 42

6. Unidad 4.2 - Programando con Lambda Cálculo. 44
6.1. Representaciones de tipos de datos. 44
6.1.1. Representación de booleanos. 44
6.1.2. Representación de pares. 45
6.1.3. Representación del tipo Either. 46
6.2. Representación de naturales. 47
6.2.1. Eliminación de naturales: foldn. 47
6.2.2. Especificación de naturales. 47
6.2.3. Solución. 47
6.2.4. Ejemplos. Representación de otros naturales. 48
6.2.5. Notación en metalenguaje: aplicación de un término varias veces. 48
6.2.6. Definición. Numerales de Church. 48
6.3. Representación de listas. 49
6.3.1. Especificación de listas. 49
6.3.2. Solución (Pares de Church para listas). 49
6.3.3. Ejemplos. 49
6.4. Formas y tipos recursivos. 50
6.4.1. Receta para representar tipos recursivos. 50
6.4.2. Limitaciones de la representación con fold. 50
6.4.3. Funciones recursivas. 51
6.4.4. Operador de punto fijo. 51
6.4.5. El combinador de punto fijo Y (combinador de Turing). 52
6.5. Resumen. 52

7. Unidad 4.3 - Lambda Cálculo con Sistemas de Tipos. 53
7.1. Sistemas de tipos. 53
7.1.1. Definición. Sistemas de tipos. 53
7.1.2. Ventajas adicional de los sistemas de tipos. 53
7.1.3. Tipos de chequeo. 53
7.1.4. Seguridad y propiedades formales. 53
7.2. Lenguaje de expresiones aritméticas y booleanas. 55
7.2.1. Sintaxis. 55
7.2.2. Reglas de tipado. 55
7.2.3. Derivación de tipado. 55
7.2.4. Reglas de semántica operacional. 55
7.2.5. Teorema. Progreso del lenguaje de expresiones aritméticas y booleanas. . . 56
7.2.6. Teorema. Preservación del lenguaje de expresiones aritméticas y booleanas. 57
7.3. Cálculo lambda simplemente tipado ($\lambda_{\rightarrow}$). 60
7.3.1. Sintaxis. 60
7.3.2. Reglas de tipado. 60
7.3.3. Recursión y normalización en lambda cálculo simplemente tipado. 61
7.3.4. Estrategia de evaluación para $\lambda_{\rightarrow}$. 61
7.3.5. Seguridad formal del $\lambda_{\rightarrow}$. 62
7.3.6. Lambda Cálculo Tipado a la Curry y a la Church. 62
7.4. Sistema T de Gödel. 63
7.4.1. Extensiones del sistema. 63
7.4.2. Reglas de tipado adicionales. 63
7.4.3. Reglas de evaluación adicionales. 63
7.4.4. Programando en Sistema T: Factorial. 64

8. Unidad 4.4 - Polimorfismo. 65
8.1. Polimorfismo. 65
8.1.1. Definición. Polimorfismo. 65

8.1.2. Polimorfismo Ad-hoc (sobrecarga). 65
8.1.3. Polimorfismo paramétrico. 65
8.2. Cálculo Lambda Polimórfico. Sistema F. 66
8.2.1. Sintaxis. 66
8.2.2. Reglas de tipado. 66
8.2.3. Reglas de semántica (evaluación). 67
8.2.4. Ejemplos. 67
8.3. Representación de tipos de datos en Sistema F. 67
8.3.1. Representación de booleanos. 67
8.3.2. Representación de naturales. 68
8.3.3. Representación de listas. 68

9. Unidad 5.1 - Abstracciones. 69
9.1. Abstracciones en programación funcional. 69
9.2. Functores. Generalización del map. 69
9.2.1. Introducción. 69
9.2.2. Definición de Functor en Haskell. 70
9.2.3. Ejemplos de Functores. 70
9.3. Functores Aplicativos. Generalización de la aplicación. 71
9.3.1. Introducción. 71
9.3.2. Definición de Functor Aplicativo en Haskell. 72
9.3.3. Ejemplos de Functores Aplicativos. 72
9.4. Mónadas. Abstracción para computaciones con efectos. 73
9.4.1. Introducción. 73
9.4.2. Mónadas como extensión de Functores Aplicativos. 73
9.4.3. Definición de la Clase Monad en Haskell. 74
9.4.4. Ejemplos de Instancias Monádicas. 74
9.4.5. Resumen. 75

10. Unidad 5.2 - Modelando efectos computacionales con mónadas. 76
10.1. Introducción. 76
10.1.1. Repaso. 76
10.1.2. Objetivos. 76
10.2. Caso 1. Manejo de errores con mónada Maybe. 76
10.2.1. Evaluador puro (sin manejo de errores, sin mónadas). 76
10.2.2. Evaluador con manejo de error (sin mónadas). 76
10.2.3. Evaluador imperativo monádico con manejo de errores. 77
10.2.4. Notación do. 77
10.2.5. Evaluador imperativo monádico con notación do. 78
10.3. Caso 2. Manejo de estado (log) con Acum y Writer. 78
10.3.1. Evaluador que cuenta operaciones (estilo manual). 78
10.3.2. Mónada Acum. 78
10.3.3. Evaluador que cuenta operaciones usando Acum. 79
10.3.4. Mónada Writer (generalización de Acum). 79
10.4. Caso 3. Lectura de entorno con mónada Reader. 80
10.4.1. Evaluador con lectura de entorno. 80
10.4.2. Mónada Reader. 80
10.4.3. Evaluador monádico con Reader. 80
10.5. Caso 4. Combinar efectos con la mónada compuesta M. 81
10.6. Conclusiones finales. 82
10.6.1. Observaciones. 82
10.6.2. Operaciones de las mónadas. 82

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.

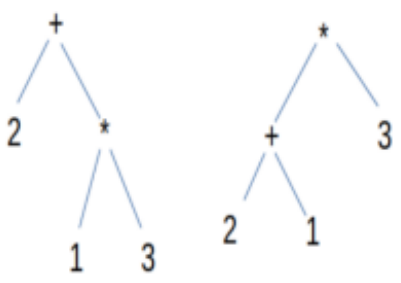
1.2.1. Ejemplo de sintaxis abstracta.

```
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` $\rightarrow$ `n` $\rightarrow$ `dn` $\rightarrow$ `dd` $\rightarrow$ `1d` $\rightarrow$ `12`
- `exp` $\rightarrow$ `exp '+' exp` $\rightarrow$ `exp '+' exp '*' exp` $\rightarrow$ `exp '+' exp '*' n` $\rightarrow \cdots \rightarrow$ `2 '+' 1 '*' 3`
- `exp` $\rightarrow$ `exp '*' exp` $\rightarrow$ `exp '+' exp '*' exp` $\rightarrow$ `exp '+' exp '*' n` $\rightarrow \cdots \rightarrow$ `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 $A \rightarrow \alpha$, 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:

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

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): $S \rightarrow aSb \mid \epsilon$

- 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)

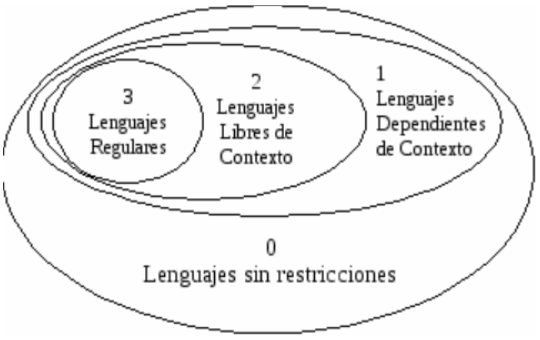
- $S \Rightarrow^* aSb$
- $S \Rightarrow^* \epsilon$
- $aSb \Rightarrow^* aaSbb$
- $aaSbb \Rightarrow^* aabb$
- $S \Rightarrow^* aaabbb$

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.**



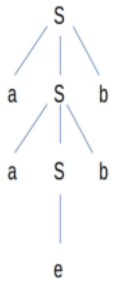
1.3.9. Definición. Árbol de parseo.

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.

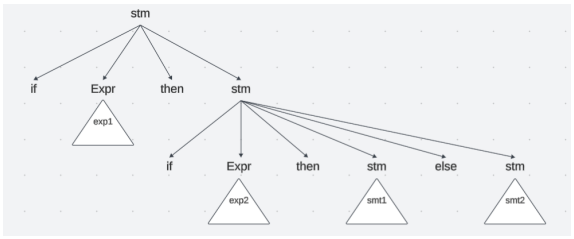
Problema de ambiguedad: *else colgado.*

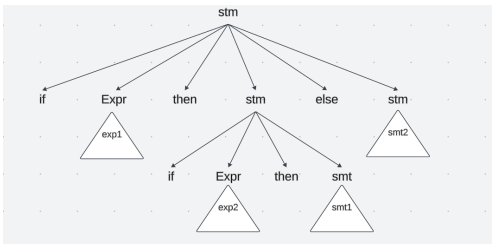
Supongamos una gramática con la siguiente regla de producción:

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

El siguiente programa tiene dos árboles de parseo:

`if` exp_1 `then if` exp_2 `then` stm_1 `else` 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` exp_1 `then (if` exp_2 `then` stm_1 `else` stm_2`)` y segundo árbol `if` exp_1 `then (if` exp_2 `then` stm_1`) else` 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)

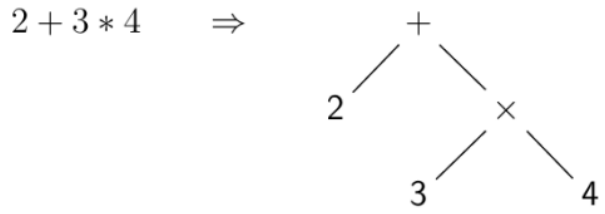
2. Unidad 2 - Analizadores Sintácticos (Parsers).

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:

```
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:

```
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.

```
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.

```
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.

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

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

```
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`).

```haskell
(<|>) :: 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.

```haskell
(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:

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

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

```haskell
(>>=) :: 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 $\leftarrow$:

```
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 `'abc'` sobre la entrada `'abcdef'`.

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 primitiva 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 ($\to$), 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 $t \to t'$ y $t \to t''$, entonces $t' = t''$.

Pasos a seguir.

1. Elegimos sobre qué derivación vamos a hacer inducción (por ejemplo, $t \to t'$).
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 $t \to t'$.

- 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 $t \to t''$ 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 $T \to t'_1$, 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 $t_1 \to t'_1$ y por lo tanto $t_2 =$ `if` t'_1 `then` t_2 `else` t_3.

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

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

  Por HI, dado que $t_1 \to t'_1$ es una subderivación de $t \to t'$ y $t_1 \to t''_1$, 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 $t \to t'$.

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 (`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:

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

- Regla para impresión de un entero:

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

5. Unidad 4.1 - 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.

La **β-reducción** es la regla que formaliza la aplicación de una función. Establece que:

$$(\lambda x.P)Q \to_\beta P[Q/x]$$

Es decir, un término con la forma $(\lambda x.P)Q$ (una función aplicada a un argumento) reduce al cuerpo de la asbtracción (P), donde la variable ligada (x) ha sido sustituida por el argumento (Q).

Definiciones.

- **β-redex:** es un término de la forma $(\lambda x.P)Q$. Esto es una expresión reducible; el término que espera ser evaluado. Su contracción es el resultado de la sustitución $P[Q/x]$.
- **β-contracción ($\to_\beta$):** si al reemplazar un β-redex en un término P por su contracción obtenemos un término P', decimos que P se β-contrae a P' y escribimos $P \to_\beta P'$.
- **β-reducción ($\to_\beta^*$):** decimos que P β-reduce a Q si $P \to_\beta^* Q$. Esto significa que Q se obtiene a partir de P mediante una secuencia (cero o más) β-contracciones. $\to_\beta^*$ es la clausura reflexiva y transitiva de $\to_\beta$.

Ejemplos.

- $(\lambda x.(\lambda y.y\ x)z)v \to_\beta ((\lambda y.y\ x)z)[x/v] \equiv ((\lambda y.y\ v)z) \to_\beta (y\ v)[z/y] \equiv z\ v$
- $(\lambda x.(\lambda y.y\ x)z)v \to_\beta ((\lambda x.(y\ x)[z/y])v) \equiv (\lambda x.z\ x)v \to_\beta (z\ x)[v/x] \equiv z\ v$
- $(\lambda x.x\ x)(\lambda x.x\ x) \to_\beta (x\ x)[(\lambda x.x\ x)/x] \equiv (\lambda x.x\ x)(\lambda x.x\ x) \to_\beta \cdots$

5.3.2. Estrategias de reducción.

Las reglas operacionales implican que, en un término complejo con múltiples β-redex, puede haber varios caminos de reducción. Las **estrategias de reducción** definen un orden para elegir qué β-redex reducir primero:

- **Reducción normal (leftmost-outermost).** Siempre reduce el β-redex más externo y a la izquierda. Esta estrategia garantiza encontrar la forma normal si existe.
- **Llamada por nombre (call-by-name).** Reduce el β-redex más externo, pero no evalúa el argumento Q antes de la sustitución.
- **Llamada por valor (call-by-value).** Primero evalúa el argumento Q hasta su forma normal (valor) antes de realizar la sustitución. Es la estrategia más común en lenguajes de programación imperativos y funcionales modernos.

5.3.3. Semántica operacional del λ-cálculo.

La semántica operacional define con precisión cómo y cúando se realizan las β-contracciones, incluso dentro de términos más grandes, mediante reglas que describen el orden de evaluación.

$$\frac{t_1 \to_\beta t_1'}{t_1\ t_2 \to_\beta t_1'\ t_2}\ \text{(E-App1)} \qquad \frac{t_2 \to_\beta t_2'}{t_1\ t_2 \to_\beta t_1\ t_2'}\ \text{(E-App2)}$$

$$\frac{t \to_\beta t'}{\lambda x.t \to_\beta \lambda x.t'}\ \text{(E-Abs)} \qquad \frac{}{(\lambda x.t_1)t_2 \to_\beta t_1[t_2/x]}\ \text{(E-AppAbs)}$$

5.3.4. Definición. Forma normal β.

Una **forma normal** β (β-nf) es el estado final de un cálculo; es un término que **no contiene β-redexes** y, por lo tanto, no puede reducirse más.

Si un término $P \to_\beta^* Q$, donde Q es una β-nf, decimos que Q es una forma normal β de P.

Por los ejemplos visto anteriormente, $z\ v$ es una β-nf del término $(\lambda x.(\lambda y.y\ x)z)v$.

5.3.5. Propiedades de la β-reducción ($\rightarrow_\beta^*$).

- **Conservación de variables libres.** La reducción nunca introduce nuevas variables libres; solo puede eliminarlas:

  $$\text{Si } P \rightarrow_\beta^* Q, \text{ entonces } FV(Q) \subseteq FV(P)$$

  Por ejemplo, $(\lambda x.(\lambda y.y))z \rightarrow_\beta (\lambda y.y)$, la variable libre z del primer término desaparece.

- **Preservación bajo sustitución.** La β-reducción se mantiene si los términos involucrados son reemplazados por sus reducciones.

  $$\text{Si } P \rightarrow_\beta^* P' \text{ y } Q \rightarrow_\beta^* Q' \text{ entonces } P[Q/x] \rightarrow_\beta^* P'[Q'/x]$$

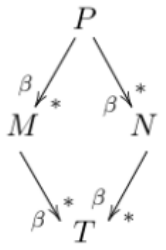
5.4. Consistencia y equivalencia en el λ-cálculo.

5.4.1. Confluencia y teorema de Church-Rosser.

Aunque la β-contracción ($\rightarrow_\beta$) es **no-determinista** (un término puede tener múltiples β-redexes y, por lo tanto, diferentes caminos de reducción), el λ-cálculo tiene una propiedad fundamental llamada **confluencia:** si partimos de un término t y aplicamos β-contracciones que llevan a distintos términos, eventualmente se llegará a la misma forma normal.

Teorema de Church-Rosser.

Si $P \rightarrow_\beta^* M$ y $P \rightarrow_\beta^* N$, entonces existe T tal que $M \rightarrow_\beta^* T$ y $N \rightarrow_\beta^* T$.



Corolario.

Esta propiedad garantiza la **unicidad** de la forma normal. Todo término tiene, a lo sumo, una única forma normal (ignorando los nombres de las variables, es decir, módulo $\equiv_\alpha$).

5.4.2. Definición. β-equivalencia.

Un término P es β-equivalente a Q (y escribimos $P =_\beta Q$) si y sólo si Q puede ser obtenido partiendo de P y realizando una serie finita de β-contracciones, β-expansiones (β-contracciones inversas) y α-conversiones.

La β-equivalencia es una relación de igualdad que la β-reducción simple, pues es **simétrica.** Además, la β-equivalencia es la noción formal de que dos términos representan la misma *función computable.*

Ejemplo. Probar que $(\lambda x\ y\ z.x\ z\ y)(\lambda x\ y.x) =_\beta (\lambda x\ y.x)(\lambda x.x)$

$$\begin{aligned}(\lambda x\ y\ z.x\ z\ y)(\lambda x\ y.x) &\to_\beta (\lambda y\ z.x\ z\ y)[(\lambda x\ y.x)/x] \\ &\equiv (\lambda y\ z.(\lambda x\ y.x)\ z\ y) \\ &\equiv (\lambda y\ z.(\lambda x(\lambda y.x))z\ y) \\ &\to_\beta (\lambda y\ z.(\lambda y.x)[z/x]\ y) \\ &\equiv (\lambda y\ z.(\lambda y.z)\ y) \\ &\to_\beta (\lambda y\ z.z[y/y]) \\ &\equiv (\lambda y\ z.z)\end{aligned}$$

Es decir, el lado izquierdo de la igualdad reduce a $P \to_\beta^* \lambda yz.z$ (esta es la β-nf, que es una función constante que devuelve el primer argumento).

$$\begin{aligned}(\lambda x\ y.x)(\lambda x.x) &\equiv (\lambda x(\lambda y.x))(\lambda x.x) \\ &\to_\beta (\lambda y.x)[(\lambda x.x)/x] \\ &\equiv (\lambda y.(\lambda x.x)) \\ &\equiv (\lambda y\ x.x)\end{aligned}$$

Y vemos que el lado derecho reduce a $Q \to_\beta^* (\lambda y\ x.x)$. Como ambos términos reducen al mismo término $T = \lambda y\ z.z$, por el teorema de Church-Rosser se concluye que:

$$(\lambda x\ y\ z.x\ z\ y)(\lambda x\ y.x) =_\beta (\lambda x\ y.x)(\lambda x.x)$$

5.4.3. Propiedades de la β-equivalencia.

- **Lema. Preservación de sustitución.** La β-equivalencia se mantiene bajo sustitución:

  Si $M =_\beta M'$ y $N =_\beta N'$, entonces $M[N/x] =_\beta M'[N'/x]$

- **Teorema de Church-Rosser para $=_\beta$.** Si dos términos son β-equivalentes, siempre se puede encontrar un término común al que ambos reducen.

  Si $P =_\beta Q$, entonces existe T tal que $P \to_\beta^* T$ y $Q \to_\beta^* T$

5.4.4. Extensionalidad.

El **principio de extensionalidad** en matemáticas establece que si dos funciones f y g producen el mismo resultado para cualquier entrada ($f(x) = g(x)$), entonces son la misma función ($f = g$).

Este principio no se cumple automáticamente en el λ-cálculo con solo β-reducción. Por ejemplo, $(\lambda x.y\ x) \not\equiv_\beta y$: la primera función toma un argumento x y se lo pasa inmediatamente a la función y; la segunda es simplemente la función y. Intuitivamente ambas funciones hacen lo mismo, actúan como la función y. Sin embargo, $\lambda x.y\ x$ tiene una λ-abstracción extra, lo que hace que la β-reducción no las iguale.

Para incluirlo se añade una nueva regla, la η-reducción.

5.4.5. Definición. η-redex, η-contracción y relación $=_{\beta\eta}$.

- **η-redex.** Llamamos **η-redex** a un término de la forma:

  $$\lambda x.Px$$

  con la condición fundamental de que la variable x no debe aparecer libre en P ($x \notin FV(P)$).

- **η-contracción.** La **η-contracción** es la operación de reducir un η-redex a su forma más simple, eliminando la abstracción innecesaria:

$$\lambda x.Px \to_\eta P$$

Esto significa que si una función $(\lambda x.Px)$ simplemente toma un argumento x y se lo aplica a una función P (donde P no depende de x), entonces esa función es trivialmente la misma que P.

Ejemplo. En el término $(\lambda x.(\lambda y.y)x)$ tenemos que $P = (\lambda y.y)$ y x no está libre en P. Aplicando la η-contracción tenemos que:

$$(\lambda x.(\lambda y.y)x) \to_\eta (\lambda y.y)$$

Ahora la función que aplica la identidad a x es η-equivalente a la función identidad.

- **Relación $=_{\beta\eta}$.** Para tener un sistema completo que capture la noción de igualdad funcional (extensionalidad) y la de computación (β-reducción), se combinan ambas reglas en una sola relación: la **$\beta\eta$-equivalencia.**

  1. Decimos que P $\beta\eta$-contrae (en un paso) a Q si y sólo si $P \to_\beta Q$ o $P \to_\eta Q$ y escribimos:

$$P \to_{\beta\eta} Q$$

  2. Llamamos $\to^*_{\beta\eta}$ a la clausura reflexiva, transitiva de $\to_{\beta\eta}$. Luego, si $P \to^*_{\beta\eta} Q$, significa que P se reduce a Q mediante una secuencia finita (cero o más) de $\beta\eta$-contracciones.
  3. Decimos que P y Q son $\beta\eta$-equivalentes, y notamos $P =_{\beta\eta} Q$, si Q se obtiene de P mediante α, β, η-conversiones (incluyendo expansiones).

- **$\beta\eta$ forma normal.** Un término que no contiene $\beta\eta$-redex es una $\beta\eta$ forma normal ($\beta\eta$-nf).

- **Teorema de Church Rosser para $=_{\beta\eta}$.** Si $P =_{\beta\eta} Q$, entonces existe T tal que:

$$P \to^*_{\beta\eta} T \qquad Q \to^*_{\beta\eta} T$$

5.5. Normalización.

5.5.1. Definición. Términos normalizantes.

- Un término P es **fuertemente normalizante** (P es SN) si toda secuencia posible de β-reducciones es finita y termina en una forma normal.
- Un término P es **débilmente normalizante** (P es WN) si tiene una forma normal. Es suficiente con que exista al menos una secuencia finita de reducción que lleve a ella.

Ejemplos.

- $\Omega = (\lambda x.x\ x)(\lambda x.x\ x)$ no es SN ni WN.
- $T_2 = (\lambda x.y)\Omega$ es SN.
- $T_3 = (\lambda x.x\ y)(\lambda x.x)$ es SN y WN.

5.5.2. Reducción normal.

La estrategia de **reducción normal** es la única que tiene la propiedad de garantizar la forma normal cuando esta existe.

- **Redex Maximal:** es un β-redex que no está contenido dentro de otro β-redex. Es el redex *más externo.*
- **Redex Maximal Izquierdo:** entre todos los redexes maximales, es el que se encuentra en la posición más a la izquierda del término.
- **Estrategia de reducción:** la reducción normal consiste en reducir siempre el β-redex maximal izquierdo en cada paso.
- **Teorema.** Si la reducción normal de un término P es infinita, entonces P no tiene forma normal.

5.5.3. Formas neutrales y normales.

Las siguientes categorías sintácticas capturan la sintaxis de los términos que no son abstracciones (na), formas normales (nf) y términos en forma neutral (neu), los cuales no son valores y no pueden reducirse.

```
na -> x | t1 t2
nf -> \x . nf | neu
neu -> x | neu nf
```

5.5.4. Reglas de evaluación para la reducción normal.

Estas reglas formalizan la estrategia de evaluación externa (no evaluar el argumento a menos que sea necesario o que la función sea neutral):

- **E-App1 (Prioridad Izquierda):** se intenta reducir la función t_1 primero.

$$\frac{\texttt{na} \rightarrow_\beta t_1'}{\texttt{na}\ t_2 \rightarrow_\beta t_1' t_2}$$

- **E-App2 (Prioridad de Argumento, solo si Función Neutral):** solo si la función es neutral (neu), se permite reducir el argumento t_2.

$$\frac{t_2 \rightarrow_\beta t_2'}{\texttt{neu}\ t_2 \rightarrow_\beta \texttt{neu}\ t_2'}$$

- **E-Abs (Eliminación en Abstracciones).**

$$\frac{t \rightarrow_\beta t'}{\lambda x.t \rightarrow_\beta \lambda x.t'}$$

- **E-AppAbs (Contracción):** se aplica el β-redex cuando es maximal y de izquierda.

$$\frac{}{(\lambda x.t_1)t_2 \rightarrow_\beta t_1[t_2/x]}$$

5.6. Resumen.

- Podemos expresar en λ-cálculo cualquier función computable, las que pueden ser calculadas por una máquina de Turing.

- Se utiliza como modelo de los lenguajes funcionales, es un modelo simple.
- La evaluación de un lambda término consiste en eliminar β-redexes aplicando β-contracciones.
- Una lambda expresión que no contiene β-redexes está en su forma normal.
- No toda expresión lambda tiene forma normal, pero si existe, es única.
- La estrategia de reducción normal nos aseguraba hallar la forma normal de expresiones que pueden reducir a una forma normal.

6. Unidad 4.2 - Programando con Lambda Cálculo.

6.1. Representaciones de tipos de datos.

El λ-cálculo, con su sintaxis y semántica mínima (basada solo en funciones y su aplicación), es suficiente para representar **todas las funciones computables.** Para *programar* con él, debemos encontrar representaciones funcionales para los tipos de datos básicos, dado que el λ-cálculo puro no tiene tipos de datos primitivos (como números o booleanos).

Para modelar un tipo de dato (naturales, booleanos, pares, etc) mediante λ-expresiones, seguimos un proceso sistemático:

1. **Valores y constructores:** establecer λ-expresiones que representen los valores del tipo y las funciones que los construyen.
2. **Operadores y eliminadores:** establecer λ-expresiones que operen o *eliminen* los valores del tipo (por ejemplo, funciones `if then else, fst, succ`).
3. **Especificación:** verificar que las λ-expresiones satisfagan el comportamiento lógico esperado (la especificación) a través de la β-equivalencia.

Observaciones.

- Dado que el λ-cálculo sin tipois es inconsistente a nivel de tipos de datos, expresiones como (`not 2`) son sintácticamente válidas, pero su comportamiento resultante no nos interesa formalmente.
- Las definiciones como `True` $\equiv (\lambda x\ y.x)$ son simplemente abreviaturas en nuestro metalenguaje para facilitar la lectura.

6.1.1. Representación de booleanos.

Queremos representar `True` y `False`, y la estructura de control **ifthenelse** (el constructor que nos permite eliminar o usar el valor booleano).

- **Especificación.** El comportamiento esperado es:

$$ifthenelse\ True\ P\ Q =_\beta P$$
$$ifthenelse\ False\ P\ Q =_\beta Q$$

- De la especificación se deduce que:

$$ifthenelse\ True =_\beta \lambda p\ q\ .\ p \text{ (una función que devuelve el primer argumento)}$$
$$ifthenelse\ False =_\beta \lambda p\ q\ .\ q \text{ (una función que devuelve el segundo argumento)}$$

- **Solución (numerales de Church para booleanos).** Adoptamos la siguiente representación, donde el booleano es una función que selecciona una de dos opciones:

$$True \equiv \lambda p\ q\ .\ p$$
$$False \equiv \lambda p\ q\ .\ q$$
$$ifthenelse \equiv \lambda x.x$$

- **Explicación.** Para que el término final sea exactamente P, necesitamos una λ-expresión que espere los argumentos P y Q y devuelva P. Por eso podemos reescribir la especificación aislando la parte que representa $True$ como:

$$ifthenelse\ True =_\beta \lambda P\ Q\ .\ P$$

La convención en el λ-cálculo (los numerales de Church para booleanos) es que el booleano es la función que realiza la selección. Por lo tanto, definimos $True$ como la función que, al recibir dos opciones, selecciona la primera:

$$True \equiv \lambda P\ Q\ .\ P$$

Si ya definimos $True$ como una función selectora de dos argumentos, $ifthenelse$ solo necesita recibir el booleano y aplicarle los dos argumentos restantes. Por lo tanto, es simplemente la función identidad sobre su primer argumento, la cual ya tiene la lógica de selección codificada:

$$ifthenelse \equiv \lambda x.x$$

- **Verificación de la especificación.** Al combinar las definiciones, debemos ver si verifican la especificación:

$$\begin{aligned} ifthenelse\ True\ P\ Q &\equiv (\lambda x.x)(\lambda p\ q\ .\ p)\ P\ Q \\ &\to_\beta (\lambda p\ q\ .\ p)\ P\ Q \text{ (1era } \beta\text{-reducción)} \\ &\equiv (\lambda p(\lambda q.p))PQ \\ &\to_\beta (\lambda q.p)[P/p]Q \\ &\equiv (\lambda q.P)Q \\ &\to_\beta P[Q/q] \equiv P \end{aligned} \tag{1}$$

- **Más operaciones sobre Booleanos.** Usando $True, False, ifthenelse$, definimos otras funciones (escribimos $ifthenelse\ P\ Q\ R$ como $if\ P\ then\ Q\ else\ R$).
  - **not:** se puede entender como una función que recibe un argumento (valor booleano); si el argumento es $True$ devuelve $False$, si es $False$ devuelve $True$.

$$\begin{aligned} not &\equiv \lambda x\ .\ if\ x\ then\ False\ else\ True \\ &\equiv \lambda x\ .\ ifthenelse\ x\ False\ True \\ &\equiv \lambda x\ .\ (\lambda x.x)\ x\ (\lambda p\ q\ .\ q)\ (\lambda p\ q\ .\ p) \\ &\to_\beta \lambda x\ .\ x\ (\lambda p\ q\ .\ q)\ (\lambda p\ q\ .\ p) \quad (\beta - \text{reducción del primer término}) \end{aligned}$$

  - **and:** se puede entender como una función que recibe dos argumentos (booleanos). Si el primero es $True$, devuelve el segundo; si el primero es $False$ directamente devuelve $False$ (cortocircuito del AND).

$$and \equiv \lambda x\ y\ .\ if\ x\ then\ y\ else\ False$$

  - **or:** se puede entender como una función que recibe dos argumentos (booleanos). Si el primero es $True$ devuelve $True$; si el primero es $False$ devuelve el segundo argumento.

$$or \equiv \lambda x\ y\ .\ if\ x\ then\ True\ else\ y$$

6.1.2. Representación de pares.

Queremos representar la tupla $pair\ P\ Q$ y las operaciones de proyección fst (primero) y snd (segundo) para *eliminar* el par.

- **Especificación.** El comportamiento esperado es:

$$\begin{aligned} fst\ (pair\ P\ Q) &=_\beta P \\ snd\ (pair\ P\ Q) &=_\beta Q \end{aligned}$$

- **Solución. Pares de Church.** Un par se representa como una función que espera un booleano para seleccionar el primer o el segundo elemento.

$$\begin{aligned} pair &\equiv \lambda x\ y\ .\ \lambda b\ .\ if\ b\ then\ x\ else\ y \\ fst &\equiv \lambda p\ .\ p\ True \\ snd &\equiv \lambda p\ .\ p\ False \end{aligned}$$

- **Explicación.** Un par se representa como una función toma dos valores (x e y) y devuelve una nueva función que espera una opción para seleccionar uno de los valores.

  El constructor `pair` es una función que toma dos elementos x e y y devuelve una función de selección ($\lambda b...$). Cuando creamos un par, por ejemplo ($pair\ A\ B$), el resultado es: $\lambda b.if\ b\ then\ A\ else\ B$. Esta función está *lista* para recibir un booleano (b) y usarlo para elegir entre A y B.

  Los eliminadores fst y snd son funciones que, al recibir un par, le inyectan el booleano correcto para forzar la selección. fst toma un par (p) y le aplica el valor $True$ (selector del primer elemento en $pair$).

- **Verificación de la especificación.** Queremos ver si con las definiciones dadas en *solución*, se verifica la especificación: $fst\ (pair\ x\ y) =_\beta x$.

$$\begin{aligned} fst\ (pair\ x\ y) &\equiv (\lambda p\ .\ p\ True)\ (pair\ x\ y) \\ &=_\beta (pair\ x\ y)\ True \\ &\equiv (\lambda b\ .\ if\ b\ then\ x\ else\ y)\ True \\ &=_\beta if\ True\ then\ x\ else\ y \\ &=_\beta x \end{aligned}$$

6.1.3. Representación del tipo Either.

Queremos representar el tipo de datos *Either* de Haskell con el eliminador *either*:

```
data Either a b = Left a | Right b

either :: Either a b -> (a -> c) -> (b -> c) -> c
either (Left a) f g = f a
either (Right b) f g = g b
```

- **Especificación.** El comportamiento esperado es:

$$\begin{aligned} either\ (Left\ a)\ f\ g &=_\beta f\ a \\ either\ (Right\ b)\ f\ g &=_\beta g\ b \end{aligned}$$

- **Solución.**

$$\begin{aligned} Left\ a &\equiv \lambda f\ g\ .\ f\ a \\ Right\ b &\equiv \lambda f\ g\ .\ g\ b \\ either &\equiv \lambda x.x \end{aligned}$$

- **Explicación.** Ambos constructores ($Left\ a$ y $Right\ b$) son funciones que esperan dos argumentos funcionales: una función f (para manejar el caso $Left$) y una función g (para manejar el caso $Right$).

  $Left\ a$ elige y aplica la primera función (f) a su valor interno (a), mientras que $Right\ b$ elige y aplica la segunda función (g) a su valor interno (b).

El eliminador, según la especificación, simplemente necesita tomar el valor $Either\ (x)$ y aplicarle las dos funciones de manejo (f y g). Por lo tanto, $either \equiv \lambda x.x$ es suficiente si asumimos la β-expansión de la especificación: $either \equiv \lambda x\ f\ g\ .\ x\ f\ g$.

- **Verificación de la especificación.** Queremos ver si con las definiciones dadas en *solución* se verifica la especificación: $either\ (Left\ a)\ f\ g =_\beta f\ a$.

$$\begin{aligned} either\ (Left\ a)\ f\ g &=_\beta (\lambda x.x)\ (Left\ a)\ f\ g \\ &\equiv (Left\ a)\ f\ g \\ &=_\beta (\lambda f'\ g'\ .\ f'\ a)\ f\ g \\ &=_\beta f\ a \end{aligned}$$

6.2. Representación de naturales.

Para representar los numeros naturales (Nat) en el λ-cálculo puro, se utiliza la **Técnica de Church** (o Numerales de Church). Esta técnica codifica un número n como una función que aplica un proceso n veces.

Los naturales son un tipo recursivo muy simple. En Haskell:

```
data Nat = Zero | Succ Nat
```

6.2.1. Eliminación de naturales: foldn.

El patrón de **eliminación** para este tipo de datos recursivos es el `fold`, que define cómo procesar el valor. En Haskell el `fold` para naturales es:

```
foldn :: Nat -> (a -> a) -> a -> a
foldn Zero s z = z
foldn (Succ n) s z = s (foldn n s z)
```

donde s es la función a aplicar repetidamente (el sucesor en nuestro ejemplo) y z es el valor inicial (cero en nuestro ejemplo). Por lo tanto, para representar los naturales necesitamos definir:

$$\begin{aligned} Zero &:: Nat \\ Succ &:: Nat \to Nat \\ foldn &:: Nat \to (a \to a) \to a \to a \end{aligned}$$

6.2.2. Especificación de naturales.

Tomando el `foldn` como el eliminador fundamental, la especificación de los naturales en el λ-cálculo es:

$$\begin{aligned} foldn\ Zero &=_\beta \lambda s\ z\ .\ z \\ foldn\ (Succ\ n) &=_\beta \lambda s\ z\ .\ s\ (foldn\ n\ s\ z) \end{aligned}$$

6.2.3. Solución.

La solución más elegante se alcanza fijando el eliminador `foldn` como la función identidad, de forma que el número n sea la propia función `fold n`. De esa forma, las definiciones para `Zero` y `Succ` se vuelven directas:

$$\begin{aligned} foldn &\equiv \lambda x.x \\ Zero &\equiv \lambda s\ z\ .\ z \\ Succ\ n &\equiv \lambda n\ .\lambda s\ z\ .\ s\ (n\ s\ z) \end{aligned}$$

6.2.4. Ejemplos. Representación de otros naturales.

Podemos ver que *uno* se representa como el sucesor de cero, mientras que *dos* es el sucesor, del sucesor de cero.

$$\begin{aligned} uno &\equiv Succ\ Zero \\ &\equiv (\lambda n\ .\lambda s\ z\ .\ s\ (n\ s\ z))\ Zero \\ &=_\beta \lambda s\ z\ .\ s\ (Zero\ s\ z) \\ &\equiv \lambda s\ z\ .\ s\ ((\lambda s\ z\ .\ z)\ s\ z) \\ &=_\beta \lambda s\ z\ .\ s\ z \end{aligned}$$

$$\begin{aligned} dos &\equiv Succ\ uno \\ &\equiv (\lambda n\ .\lambda s\ z\ .\ s\ (n\ s\ z))\ uno \\ &=_\beta \lambda s\ z\ .\ s\ (uno\ s\ z) \\ &=_\beta \lambda s\ z\ .\ s\ ((\lambda s\ z\ .\ s\ z)\ s\ z) \\ &=_\beta \lambda s\ z\ .\ s\ (s\ z) \end{aligned}$$

6.2.5. Notación en metalenguaje: aplicación de un término varias veces.

Queremos representar n aplicaciones de un término. En nuestro metalenguaje anotaremos:

$$\begin{aligned} F^0 M &= M \\ F^{n+1} M &= F^n (F\ M) \end{aligned}$$

Ejemplo:

$$\begin{aligned} (\lambda x\ y\ .\ x)^3\ z &\equiv (\lambda x\ y\ .\ x)^2\ ((\lambda x\ y\ .\ x) z) \\ &\equiv (\lambda x\ y\ .\ x)^1 ((\lambda x\ y\ .\ x)((\lambda x\ y\ .\ x)\ z)) \\ &\equiv (\lambda x\ y\ .\ x)^0 ((\lambda x\ y\ .\ x)\ ((\lambda x\ y\ .\ x)((\lambda x\ y\ .\ x)\ z))) \\ &\equiv (\lambda x\ y\ .\ x)\ ((\lambda x\ y\ .\ x)((\lambda x\ y\ .\ x)\ z)) \end{aligned}$$

6.2.6. Definición. Numerales de Church.

Para cada $n \in \mathbb{N}$, el **numeral de Church** para n es un término $\underline{n}$ definido como:

$$\underline{n} \equiv \lambda f\ x\ .\ f^n\ x$$

Observación. Notar que $\underline{0} \equiv False$. No importa, ya que no usamos tipos. La especificación sólo dice que hacer en caso que los argumentos tengan la forma correcta.

Funciones sobre numerales de Church.

- **Suma.** La suma de dos números $(n + m)$ es simplemente aplicar la función $Succ\ m$ veces a n. La especificación de la suma es:

$$\begin{aligned} suma\ \underline{n}\ \underline{0} &=_\beta \underline{n} \\ suma\ \underline{n}\ (Succ\ m) &=_\beta Succ\ (suma\ \underline{n}\ \underline{m}) \end{aligned}$$

  O equivalentemente, usando foldn, como aplicar la función sucesor m veces a n:

$$suma\ \underline{n}\ \underline{m} =_\beta foldn\ \underline{m}\ Succ\ \underline{n}$$

Osea que podemos definir *suma* en λ-cálculo como:

$$suma \equiv \lambda n\ m\ .\ m\ Succ\ n$$

- **Producto.** ...
- **izZero.** ...

6.3. Representación de listas.

Generalizamos los numerales de Church a listas. Las listas están dadas por el siguiente tipo de datos recursivo:

```
data List a = Nil | Cons a (List a)
```

El eliminador estándar para listas es $foldr$, que define el patrón de consumo:

```
foldr :: List a -> (a -> b -> b) -> b -> b
foldr Nil c n = n
foldr (Cons x xs) c n = c x (foldr xs c n)
```

Por lo tanto, para representar listas necesitamos definir:

```
Nil :: List a
Cons :: a -> List a -> List a
foldr :: List a -> (a -> b -> b) -> b -> b
```

6.3.1. Especificación de listas.

Tomando el $foldr$ como el eliminador fundamental, la especificación de las listas en el λ-cálculo es:

$$foldr\ Nil =_\beta \lambda c\ n\ .\ n$$
$$foldr\ (Cons\ x\ xs) =_\beta \lambda c\ n\ .\ c\ x\ (foldr\ xs\ c\ n)$$

6.3.2. Solución (Pares de Church para listas).

Al igual que con los numerales, la solución se alcanza fijando el eliminador $foldr$ como la función identidad ($\lambda x.x$). Esto implica que la lista L es la función $foldr\ L$:

$$\begin{aligned} foldr &\equiv \lambda x.x \\ Nil &\equiv \lambda c\ n\ .\ n \\ Cons &\equiv \lambda x\ xs\ .\ \lambda c\ n\ .\ c\ x\ (xs\ c\ n) \end{aligned}$$

6.3.3. Ejemplos.

- Lista vacía: $Nil \equiv \lambda c\ n\ .\ n$
- Cons: $Cons \equiv \lambda x\ xs\ .\ \lambda c\ n\ .\ c\ x\ (xs\ c\ n)$
- Lista con el número 3.

$$\begin{aligned} [3] &\equiv Cons\ 3\ Nil \\ &\equiv (\lambda x\ xs\ .\ \lambda c\ n\ .\ c\ x\ (xs\ c\ n))\ 3\ Nil \\ &=_\beta \lambda c\ n\ .\ c\ 3\ (Nil\ c\ n) \\ &\equiv \lambda c\ n\ .\ c\ 3\ ((\lambda c\ n\ .\ n)\ c\ n) \\ &=_\beta \lambda c\ n\ .\ c\ 3\ n \end{aligned}$$

- Lista con los números 2 y 3.

$$\begin{aligned}[2,3] &\equiv Cons\ 2\ (Cons\ 3\ Nil)\\ &\equiv (\lambda\ x\ xs\ .\ \lambda c\ n\ .\ c\ x\ (xs\ c\ n))\ 2\ (Cons\ 3\ Nil)\\ &=_\beta \lambda c\ n\ .\ c\ 2\ ((Cons\ 3\ Nil)\ c\ n)\\ &=_\beta \lambda c\ n\ .\ c\ 2\ ((\lambda c\ n\ .\ c\ 3\ n)\ c\ n)\\ &=_\beta \lambda c\ n\ .\ c\ 2\ (c\ 3\ n)\end{aligned}$$

- Función *length*, recursivamente en Haskell tiene la siguiente forma:

```
length :: List a -> Nat
length Nil = Zero
length (Cons x xs) = Succ (length xs)
```

  Si la reescribimos usando $foldr$:

$$length\ xs = foldr\ xs\ (\lambda x\ n\ .\ Succ\ n)\ Zero$$

  Y por lo tanto, en λ-cálculo:

$$length \equiv \lambda xs\ .\ xs\ (\lambda x\ n\ .\ Succ\ n)\ Zero$$

6.4. Formas y tipos recursivos.

6.4.1. Receta para representar tipos recursivos.

El método utilizado para los naturales y las listas es una receta general para codificar cualquier tipo de datos recursivo en el λ-cálculo.

1. **Identificar Constructores:** determinar las formas de construir el tipo de datos (por ejemplo, $Nil, Cons$).
2. **Escribir el fold correspondiente:** definir la función de eliminación canónica para el tipo (por ejemplo, $foldn, foldr$).
3. **Especificación:** usar las ecuaciones del $fold$ como la especificación de comportamiento deseado en el λ-cálculo.
4. **Definir λ-términos:** definir el $fold$ como la función identidad y derivar a partir de las ecuaciones los λ-términos correspondientes a los constructores.

6.4.2. Limitaciones de la representación con fold.

La técnica de representar tipos recursivos mediante su patrón de eliminación ($foldn, foldr$) es muy potente, pero tiene limitaciones cuandose se intenta definir funciones que requieren acceso simultáneo al valor actual y al **valor previo** de la recursión. Las siguientes funciones, por ejemplo, son díficiles de representar:

- **Predecesor de naturales (pred):** para calcular $pred\ (Succ\ n)$, se necesita saber el valor de n, no solo el resultado del $fold$ de n.
- **Cola de una lista (tail):** similarmente, para obtener la cola, se necesita información sobre la lista *interna* que se está procesando.

La solución es usar **tuplamiento** (accareo de información). Para definir estas funciones usando $fold$, es necesario que el resultado de la función sea una tupla (*pair*) que *acarree* la información necesaria para el siguiente paso. El resultado deseado se extrae luego de una de las componentes de la tupla (usando fst o snd).

Ejemplo. Función predecesor (pred).

Para definir $pred$ usando $foldn$, se define una función auxiliar $pred'$ que devuelve una dupla (predecesor, n):

$$\begin{aligned} pred'\ Zero &= (0,0) \\ pred'\ (Succ\ n) &= (snd\ (pred'\ n), Succ\ (snd\ (pred'\ n))) \\ pred\ n &= fst\ (pred'\ n) \end{aligned}$$

Y luego:

$$pred' = \lambda n\ .\ foldn\ n\ (\lambda pn\ .\ (snd\ pn, Succ\ (snd\ pn)))\ (0,0)$$

6.4.3. Funciones recursivas.

El λ-cálculo, siendo un lenguaje de programación universal, debe permitir la definición de **funciones recursivas.**

Por ejemplo, queremos definir la función **factorial** ($fact$):

$$fact \equiv \lambda n\ .\ if\ (isZero\ n)\ then\ \underline{1}\ else\ prod\ n\ (fact\ (pred\ n))$$

Pero esto no es una definición valida en el λ-cálculo puro! El término $fact$ aparece en su propio lado derecho antes de ser completamente definido, violando la regla de sustitución. Las funciones en el λ-cálculo deben ser anónimas y no autocontenidas.

Transformando la recursión en un punto fijo. Para resolver esto, se abstrae la llamada recursiva problemática.

- **Abstracción.** Se reemplaza la llamada recursiva $fact$ por una variable f. Esto crea el *cuerpo* de la función recursiva, que toma la función recursiva como primer argumento:

  $$B \equiv \lambda f\ n\ .\ if\ (isZero\ n)\ then\ \underline{1}\ else\ prod\ n\ (f\ (pred\ n))$$

- **Ecuación de punto fijo.** La definición original se convierte en una ecuación: la función $fact$ es el resultado de aplicar el cuerpo B (que toma por argumento la función f) a $fact$ mismo:

  $$fact =_\beta B\ fact$$

  Luego definir $fact$ es resolver esta ecuación en la incógnita $fact$.

6.4.4. Operador de punto fijo.

Para resolver ecuaciones de la forma $X =_\beta BX$ se utiliza un operador de punto fijo **F**. Un **operador de punto fijo** es un término **F** tal que:

$$\mathbf{F}\ B =_\beta B\ (\mathbf{F}\ B)$$

Si encontramos tal operador, podemos definir nuestra función recursiva $fact$:

$$\begin{aligned} fact &\equiv \mathbf{F}\ B \\ &=_\beta B\ (\mathbf{F}\ B) \\ &\equiv B\ fact \end{aligned}$$

Es decir que $fact$ es:

$$fact \equiv \mathbf{F}B \equiv \mathbf{F}\ (\lambda f\ n\ .\ if\ (isZero\ n)\ then\ \underline{1}\ else\ prod\ n\ (f\ (pred\ n)))$$

6.4.5. El combinador de punto fijo Y (combinador de Turing).

El λ-cálculo es lo suficientemente potente como para definir operadores de punto fijo. El más conocido es el **Combinador Y** (o Combinador de Punto Fijo de Turing):

$$\mathbf{Y} = \lambda x\ .(\lambda y\ .\ x\ (y\ y))\ (\lambda y\ .\ x\ (y\ y))$$

Teorema. El combinador $\mathbf{Y}$ es universal, es decir, todo término X tiene punto fijo dado por $(Y\ X)$:

$$\mathbf{Y}\ X =_\beta X\ (\mathbf{Y}\ X)$$

El operador $\mathbf{Y}$ no es el único operador de punto fijo. Luego, la existencia del combinador $\mathbf{Y}$ es lo que demuestra formalmente que el λ-cálculo es capaz de expresar **toda función recursiva,** lo que confirma su poder universal de computación.

6.5. Resumen.

El λ-cálculo es un **lenguaje de programación completo** (Turing completo) que logra esta potencia con una sintaxis mínima:

- **Representación de datos:** los tipos de datos (Booleanos, Pares, Naturales, Listas) se representan como **funciones** que codifican su patrón de uso ($fold$).
- **Recursión:** las funciones recursivas se definen utilizando el **Operador de Punto Fijo Y,** lo que permite que una función se refiera a sí misma sin violar las reglas de λ-abstracción.

7. Unidad 4.3 - Lambda Cálculo con Sistemas de Tipos.

7.1. Sistemas de tipos.

7.1.1. Definición. Sistemas de tipos.

Los **sistemas de tipos** forman la base teórica de la seguridad y el diseño de la mayoría de los lenguajes de programación modernos. Su objetivo principal es garantizar la ausencia de ciertos errores antes de que el programa se ejecute.

Según Benjamin Pierce, un sistema de tipos es: *un método sintáctico para probar la ausencia de ciertos comportamientos mediante la clasificación de frases de acuerdo a los valores que computan.*

7.1.2. Ventajas adicional de los sistemas de tipos.

Más allá de la detección temprana de errores, los sistemas de tipos ofrecen beneficios cruciales:

- **Especificación rudimentaria o documentación.** Las firmas de tipos actúan como una especificación de la función (qué recibe y qué devuelve), que siempre está actualizada y verificada por el compilador.
- **Abstracción.** Los tipos permiten definir interfaces claras e independientes de la implementación.
- **Optimización.** La información de tipos permite al compilador generar código máquina más eficiente.
- **Lenguajes seguros.** Son esenciales para garantizar que el lenguaje proteja sus abstracciones.

7.1.3. Tipos de chequeo.

1. **Chequeo estático (tiempo de compilación).** Los sistemas de tipos proveen un **chequeo estático,** es decir, en tiempo de compilación. Al ser estático sólo se puede garantizar la ausencia de ciertos errores.

   Los sistemas de tipos son necesariamente **conservadores,** permiten probar la ausencia de determinados errores, pero no probar la presencia de errores. El siguiente ejemplo es usualmente rechazado aunque `test` sea siempre verdadero:

   ```
   if test then S else (error de tipo)
   ```

   Los lenguajes con **tipos dependientes** permiten hacer algunas de estas verificaciones estáticamente.

2. **Chequeo dinámico.** El chequeo del tipo de un programa es realizado en tiempo de **ejecución.** Por ejemplo, en una expresión de la forma $e_1 + e_2$, primero se evalúan las expresiones y dependiendo del tipo de éstas, se aplica la operación $+$.

   Algunos lenguajes incluyen un chequeo dinámico, incluso teniendo un sistema de tipos para el chequeo estático. Los lenguajes que incluyen un chequeo dinámico, pero no un sistema de tipos se denominan **lenguajes con tipado dinámico.**

7.1.4. Seguridad y propiedades formales.

Lenguaje Seguro. Según Pierce, *un lenguaje es seguro si protege sus abstracciones.* Por ejemplo:

- Si un lenguaje provee la abstracción de arreglo (array) para ser seguro debe garantizar que no hay manera de escribir fuera del arreglo.
- Un lenguaje con variables locales deben poder accederse sólo bajo su alcance (scope).

- En lenguajes no seguros (C, C++), el uso incorrecto de punteros puede llevar a un comportamiento impredecible o a violaciones de memoria.

Seguridad y tipado.

Un lenguaje **seguro** no es igual a **tipado estático.** Formalizaremos una propiedad de seguridad básica: *Los términos tipados no causan errores.*

Las formas normales que no son valores se denominan **términos atascados.** El sistema de tipos debe garantizar que los términos bien tipados nunca se atasquen.

Seguridad = Progreso + Preservación.

- **Progreso.** Si un término t tiene tipo T ($t : T$), entonces o t es un **valor** (no se puede reducir más) o existe un término t' al que t puede reducir ($t \to t'$).

  ($\Rightarrow$) Esta propiedad implica que un término tipado nunca se atasca.

- **Preservación.** Si un término t tiene tipo T ($t : T$) y se reduce a t' ($t \to t'$), entonces t' también tiene tipo T ($t' : T$).

  ($\Rightarrow$) Esta propiedad implica que la evaluación (reducción) preserva el tipo. Es decir, un término bien tipado nunca se convierte en un término mal tipado durante el cálculo.

7.2. Lenguaje de expresiones aritméticas y booleanas.

Este sistema formaliza un lenguaje de programación mínimo que combina lógica booleana y aritmética simple, permitiendo introducir las nociones de tipos y seguridad del lenguaje.

7.2.1. Sintaxis.

```
t -> true | false | if t then t else t | 0 | suc t | pred t | iszero t
v -> true | false | nv
nv -> 0 | suc nv
T -> Bool | Nat
```

- t: las expresiones posibles en el lenguaje.
- v: términos que no pueden reducirse más.
- nv: representación de los naturales (Numerales de Church codificados).
- T: los tipos posibles en el lenguaje.

7.2.2. Reglas de tipado.

Agregamos la **relación de tipado** $(t : T)$, que se lee como **el término t tiene tipo T.**

Decimos que un término es **tipado** si existe al menos un tipo T tal que $t : T$. Esta relación se define como la menor relación que satisface las siguientes reglas de inferencia:

- $\overline{true : Bool}$ (T-True)
- $\overline{false : Bool}$ (T-False)
- $\dfrac{t_1 : Bool \quad t_2 : T \quad t_3 : T}{if\ t_1\ then\ t_2\ else\ t_3\ :\ T}$ (T-If)
- $\overline{0 : Nat}$ (T-Zero)
- $\dfrac{t : Nat}{pred\ t : Nat}$ (T-Pred)
- $\dfrac{t : Nat}{suc\ t : Nat}$ (T-Suc)
- $\dfrac{t : Nat}{iszero\ t : Bool}$ (T-IsZero)

7.2.3. Derivación de tipado.

La prueba de que un término tiene un tipo específico se realiza construyendo un **árbol de derivación** usando estas reglas, desde la raíz hacia las hojas:

Por ejemplo, probar que el término `if iszero 0 then 0 else 0 : Nat` se realiza:

$$\dfrac{\dfrac{\dfrac{\overline{\texttt{0 : Nat}}\ \text{T-Zero}}{\texttt{iszero 0 : Bool}}\ \text{T-IsZero} \quad \dfrac{}{\texttt{0 : Nat}}\ \text{T-Zero} \quad \dfrac{\overline{\texttt{0 : Nat}}\ \text{T-Zero}}{\texttt{pred 0 : Nat}}\ \text{T-Pred}}{\texttt{if iszero 0 then 0 else pred 0 : Nat}}\ \text{T-If}$$

7.2.4. Reglas de semántica operacional.

La semántica (cómo se evalúan los términos) se define por la **relación de reducción** $\rightarrow \in T \times T$, que describe un único paso de ejecución. Esta definida por las siguientes reglas:

Reglas para lógica booleana.

- $\overline{if\ true\ then\ t_2\ else\ t_3 \to t_2}$ (E-IfTrue)
- $\overline{if\ false\ then\ t_2\ else\ t_3 \to t_2}$ (E-IfFalse)
- $\dfrac{t_1 \to t_1'}{if\ t_1\ then\ t_2\ else\ t_3 \to if\ t_1'\ then\ t_2\ else\ t_3}$ (E-If)

Reglas para la aritmética.

- $\dfrac{t \to t'}{suc\ t \to suc\ t'}$ (E-Suc)
- $\overline{pred\ 0 \to 0}$ (E-PredZero)
- $\overline{pred\ (suc\ nv) \to nv}$ (E-PredSuc)
- $\dfrac{t \to t'}{pred\ t \to pred\ t'}$ (E-Pred)
- $\overline{iszero\ 0 \to true}$ (E-IsZeroZero)
- $\overline{iszero\ (suc\ nv) \to false}$ (E-IsZeroSuc)
- $\dfrac{t \to t'}{iszero\ t \to iszero\ t'}$ (E-IsZero)

7.2.5. Teorema. Progreso del lenguaje de expresiones aritméticas y booleanas.

Si $t : T$ entonces t es un valor o existe t' tal que $t \to t'$.

Demostración. Por inducción en la derivación de tipado $\mathcal{D}$ para $t : T$. La prueba procede analizando la última regla aplicada en la derivación $\mathcal{D}$.

- **Caso (T-True).** La derivación $\mathcal{D}$ termina con $\overline{true : Bool}$. Como el término $t = true$ y $true$ es un valor (categoría sintáctica v), entonces se cumple la primera parte del teorema.
- **Caso (T-False).** La derivación $\mathcal{D}$ termina con $\overline{false : Bool}$. Como el término $t = false$ y $false$ es un valor (categoría sintáctica v), entonces se cumple la primera parte del teorema.
- **Caso (T-Zero).** La derivación $\mathcal{D}$ termina con $\overline{0 : Nat}$. Como el término $t = 0$ y 0 es un valor (categoría sintáctica nv), entonces se cumple la primera parte del teorema.
- **Caso (T-Suc).** La derivación es $t = suc\ t_1$ con premisa $t_1 : Nat$.
  - **Subcaso t_1 es valor.** Como t_1 es un valor y además $t_1 : Nat$, entonces t_1 debe ser un valor numérico (categoría sintáctica nv).

    Luego, $t = suc\ nv$, y $suc\ nv$ es un término que pertenece a la categoría sintáctica nv. Por lo tanto, t es un valor.
  - **Subcaso t_1 no es un valor.** Por HI sobre t_1 vale el teorema, es decir, existe t_1' tal que $t_1 \to t_1'$.

    Luego, podemos aplicar la regla de semántica (E-Suc) y tenemos que a partir de $t_1 \to t_1'$, entonces $t = suc\ t_1 \to suc\ t_1' = t'$. Por lo tanto, existe un paso de reducción tal que $t \to t'$.
- **Caso (T-Pred).** La derivación es $t = pred\ t_1$, con premisa $t_1 : Nat$.
  - **Subcaso t_1 es un valor tal que $t_1 = 0$.** Como $t_1 = 0$, entonces la regla es $t = pred\ 0$.

Podemos aplicar la regla semántica (E-PredZero) y tenemos que $pred\ 0 \to 0$. Luego, concluimos que existe un paso de reducción tal que $t \to 0$.

- **Subcaso t_1 es un valor tal que $t_1 = suc\ nv$.** Como $t_1 = suc\ nv$, entonces la regla es $t = pred\ (suc\ nv)$.

  Podemos aplicar la regla semántica (E-PredSuc) y tenemos que $pred\ (suc\ nv) \to nv$. Luego, concluimos que existe un paso de reducción tal que $t \to nv$.

- **Subcaso t_1 no es un valor.** Por HI sobre t_1, existe t_1' tal que $t_1 \to t_1'$.

  Luego, podemos aplicar la regla semántica (E-Pred) y tenemos que a partir de $t_1 \to t_1'$, entonces $t = pred\ t_1 \to pred\ t_1' = t'$. Por lo tanto, existe un paso de reducción tal que $t \to t'$.

- **Caso (T-IsZero).** La derivación es $t = iszero\ t_1$, con premisa $t_1 : Nat$.

  - **Subcaso t_1 es un valor tal que $t_1 = 0$.** Como $t_1 = 0$, entonces la regla es $t = iszero\ 0$.

    Podemos aplicar la regla (E-IsZeroZero) y tenemos que $iszero\ 0 \to true$. Luego, concluimos que existe un paso de reducción tal que $t \to true$.

  - **Subcaso t_1 es un valor tal que $t_1 = suc\ nv$.** Como $t_1 = suc\ nv$, entonces la regla es $t = iszero\ (suc\ nv)$.

    Luego, podemos aplicar la regla semántica (E-IsZeroSuc) y tenemos que $iszero\ (suc\ nv) \to false$. Luego, concluimos que existe un paso de reducción tal que $t \to false$.

  - **Subcaso t_1 no es un valor.** Por HI sobre t_1, existe t_1' tal que $t_1 \to t_1'$.

    Luego, podemos aplicar la regla semántica (E-IsZero) y tenemos que a partir de $t_1 \to t_1'$, entonces $t = iszero\ t_1 \to iszero\ t_1' = t'$. Por lo tanto, existe un paso de reducción tal que $t \to t'$.

- **Caso (T-If).** La derivación es $t = if\ t_1\ then\ t_2\ else\ t_3$, con premisas $t_1 : Bool, t_2 : T, t_3 : T$.

  - **Subcaso t_1 es un valor.** Como $t_1 : Bool$, entonces t_1 debe ser $true$ o $false$.

    Si $t_1 = true$, podemos aplicar (E-IfTrue) y surge que $t \to t_2$.

    Si $t_1 = false$, podemos aplicar (E-IfFalse) y surge que $t \to t_3$.

    Conclusión, existe un paso de reducción $t \to t'$.

  - **Subcaso t_1 no es un valor.** Por HI sobre t_1, existe t_1' tal que $t_1 \to t_1'$.

    Luego, aplicando la regla semántica (E-If) tenemos que $t = if\ t_1\ then\ t_2\ else\ t_3 \to if\ t_1'\ then\ t_2\ else\ t_3 = t'$. Por lo tanto, existe un paso de reducción tal que $t \to t'$.

Dado que todos los casos han sido cubiertos, el teorema de progreso queda demostrado. □

7.2.6. Teorema. Preservación del lenguaje de expresiones aritméticas y booleanas.

Si $t : T$ y $t \to t'$, entonces $t' : T$.

Demostración. La prueba se realiza por inducción sobre la derivación de **reducción** $t \to t'$ (observar que no en las reglas de tipado), analizando la última regla de semántica operacional aplicada. En cada caso, debemos mostrar que existe un árbol de derivación para $t' : T$.

- **Caso (E-IfTrue).** Si se aplicó esta regla, entonces tenemos que $t = if\ true\ then\ t_2\ else\ t_3 \to t_2 = t'$.

  Ahora veamos por reglas de tipado, que para tipar el término t se tuvo que haber aplicado la regla (T-If). Por hipótesis de este teorema tenemos que $t : T$, luego la derivación debe haber

sido:

$$\frac{\overline{true : Bool} \qquad t_2 : T \qquad t_3 : T}{if\ true\ then\ t_2\ else\ t_3 : T}$$

Por las premisas de la regla de tipado, tenemos que $t_2 : T$. Luego, dado que $t : T$ y $t \to t_2 = t'$ con $t' : T$, se cumple el teorema.

- **Caso (E-IfFalse).** Si se aplicó esta regla, entonces tenemos que $t = if\ false\ then\ t_2\ else\ t_3 \to t_3 = t'$.

  Ahora veamos por reglas de tipado, que para tipar el término t se tuvo que haber aplicado la regla (T-If). Por hipótesis de este teorema tenemos que $t : T$, luego la derivación debe haber sido:

$$\frac{\overline{false : Bool} \qquad t_2 : T \qquad t_3 : T}{if\ false\ then\ t_2\ else\ t_3 : T}$$

  Por las premisas de la regla de tipado, tenemos que $t_3 : T$. Luego, dado que $t : T$ y $t \to t_3 = t'$ con $t' : T$, se cumple el teorema.

- **Caso (E-PredZero).** Si se aplicó esta regla, entonces tenemos que $t = pred\ 0 \to 0 = t'$.

  Ahora veamos por reglas de tipado, que para tipar el término t se tuvo que haber aplicado la regla (T-Pred). Por hipótesis de este teorema tenemos que $t : T$ y como $t = pred\ 0$, el tipo T debe ser Nat.

  Como además $0 : Nat$ por la regla (T-Zero), entonces tenemos que $t = pred\ 0 : Nat$ y $t \to 0 = t' : Nat$. Por lo tanto, se cumple el teorema.

- **Caso (E-PredSucc).** Si se aplicó esta regla, entonces tenemos que $t = pred\ (suc\ nv) \to nv = t'$ (donde nv es un valor numérico).

  Ahora veamos por reglas de tipado, que para tipar el término t se tuvo que haber aplicado la regla (T-Pred). Por hipótesis de este teorema tenemos que $t : T$ y como $t = pred\ (suc\ nv)$, el tipo T debe ser Nat.

  Como además $suc\ nv : Nat$ por la regla (T-Suc) entocnes nv tiene tipo Nat y luego tenemos que $t = pred\ (suc\ nv) : Nat$ y $t \to nv = t' : Nat$. Por lo tanto, se cumple el teorema.

- **Caso (E-IsZeroZero).** Si se aplicó esta regla, entonces tenemos que $t = iszero\ 0 \to true = t'$.

  Ahora veamos por reglas de tipado, que para tipar el término t se tuvo que haber aplicado la regla (T-IsZero). Por hipótesis de este teorema tenemos que $t : T$ y como $t = iszero\ 0$, el tipo T debe ser bool y $0 : Nat$.

  Como además $true : Bool$ por regla (T-True), entonces tenemos que $t : Bool$ y $t \to t' : Bool$. Por lo tanto, se cumple el teorema.

- **Caso (E-IsZeroSuc).** Si se aplicó esta regla, entonces tenemos que $t = iszero\ (suc\ nv) \to false = t'$.

  Ahora veamos por reglas de tipado, que para tipar el término t se tuvo que haber aplicado la regla (T-IsZero). Por hipótesis de este teorema tenemos que $t : T$ y como $t = iszero\ (suc\ nv)$, el tipo T debe ser bool y $(suc\ nv) : Nat$ por premisa de la regla.

  Como además $false : Bool$ por regla (T-False), entonces tenemos que $t : Bool$ y $t \to t' : Bool$. Por lo tanto, se cumple el teorema.

- **Caso (E-If).** Si se aplicó esta regla entonces tenemos que $t = if\ t_1\ then\ t_2\ else\ t_3 \to if\ t_1'\ then\ t_2\ else\ t_3 = t'$, con la premisa $t_1 \to t_1'$.

Para tipar el término $t : T$ se tuvo que haber usado la regla (T-If): $t_1 : Bool, t_2 : T, t_3 : T$.

Ahora podemos aplicar la HI sobre el término t_1 que es *menor* que t: si $t_1 : Bool$ y $t_1 \to t'_1$, entonces $t'_1 : Bool$. Luego, podemos construir la derivación para t':

$$\frac{t'_1 : Bool \qquad t_2 : T \qquad t_3 : T}{if\ t'_1\ then\ t_2\ else\ t_3 : T} \qquad \text{(T-If)}$$

Entonces tenemos que $t : T$ y $t \to t' : T$, por lo tanto, se cumple el teorema.

- **Caso (E-Suc).** Si se aplicó esta regla entonces tenemos que $t = suc\ t_1 \to suc\ t'_1 = t'$, con la premisa $t_1 \to t'_1$.

  Para tipar el término $t : T$ se tuvo que haber usado la regla (T-Suc): $t_1 : Nat$ y el tipo T debe ser Nat.

  Ahora podemos aplicar la HI sobre el término t_1 que es *menor* que t: si $t_1 : Nat$ y $t_1 \to t'_1$, entonces $t'_1 : Nat$. Luego, podemos construir la derivación para t':

$$\frac{t'_1 : Nat}{suc\ t'_1 : Nat} \qquad \text{(T-Suc)}$$

  Entonces tenemos que $t : Nat$ y $t \to t : Nat$, por lo tanto se cumple el teorema.

- **Caso (E-Pred).** Si se aplicó esta regla entonces tenemos que $t = pred\ t_1 \to pred\ t'_1 = t'$, con la premisa $t_1 \to t'_1$.

  Para tipar el término $t : T$ se tuvo que haber usado la regla (T-Pred): $t_1 : Nat$ y el tipo T debe ser Nat.

  Ahora podemos aplicar la HI sobre el término t_1 que es *menor* que t : si $t_1 : Nat$ y $t_1 \to t'_1$, entonces $t'_1 : Nat$. Luego, podemos construir la derivación para t':

$$\frac{t'_1 : Nat}{pred\ t'_1 : Nat} \qquad \text{(T-Pred)}$$

  Entonces tenemos que $t : Nat$ y $t \to t' : Nat$, por lo tanto se cumple el teorema.

- **Caso (E-IsZero).** Si se aplicó esta regla entonces tenemos que $t = iszero\ t_1 \to iszero\ t'_1 = t'$, con la premisa $t_1 \to t'_1$.

  Para tipar el término $t : T$ se tuvo que haber usado la regla (T-IsZero): $t_1 : Nat$ y el tipo T debe ser $Bool$.

  Ahora podemos aplicar la HI sobre el término t_1 que es *menor* que t: si $t_1 : Nat$ y $t_1 \to t'_1$, entonces $t'_1 : Nat$. Luego, podemos construir la derivación para t':

$$\frac{t'_1 : Nat}{iszero\ t'_1 : Bool} \qquad \text{(T-IsZero)}$$

  Entonces tenemos que $t : Bool$ y $t \to t' : Bool$, por lo tanto se cumple el teorema.

Dado que todos los posibles pasos de reducción preservan el tipo del término, el Teorema de Preservación queda demostrado. □

7.3. Cálculo lambda simplemente tipado ($\lambda_{\to}$).

El λ-cálculo simplemente tipado ($\lambda_{\to}$) extiende el λ-cálculo puro (sin tipos) al imponer un sistema de tipos estático. Esto garantiza propiedades de seguridad y normalización, aunque restringe la expresividad (por ejemplo, al prohibir la recursión sin operadores adicionales).

7.3.1. Sintaxis.

`t := x | c | t t |λx . t`

`T := B | T → T`

- t es un conjunto de términos que incluyen variables, c que es un conjunto de constantes (tipos base, operadores, etc), aplicación y abstracción
- T son los tipos, donde B es un conjunto de tipos bases (Nat, Bool) y $T \to T$ es el tipo de una función que recibe un argumento del primer tipo T y devuelve un resultado del segundo tipo T.

7.3.2. Reglas de tipado.

Al introducir variables al lenguaje, la relación de tipado deberá tener un elemento más: un **contexto de tipado** o **entorno de tipo.**

7.3.2.1. Contexto de tipado.

Un **contexto de tipos** (Γ) es una secuencia de variables con sus tipos, actuando como un mapa de variables a tipos.

- Notación: escribimos $x : T \in \Gamma$ o $\Gamma(x) = T$ para indicar que la variable x tiene tipo T en el entorno Γ.
- Extensión: el operador coma (,) extiende un entorno: $\Gamma, x : T_1$ agrega la variable x con tipo T_1 al entorno Γ.
- Convención: se asume que los nombres de las variables ligadas se pueden renombrar (α-conversión) para evitar conflictos en el entorno. Además, el entorno vacío se denota con $\emptyset$ o simplemente se omite.

7.3.2.2. Reglas de tipado.

La **relación de tipado** es ternaria: $\Gamma \vdash t : T$. Esto se lee como: 'en el entorno Γ, el término t tiene tipo T'. A continuación introducimos las reglas de tipado para este lenguaje.

- $$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$
- $$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \quad \text{(T-App)}$$
- $$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2} \quad \text{(T-Abs)}$$

7.3.2.3. Ejemplos. Determinar el tipo de los siguientes términos dando el árbol de derivación siendo $\Gamma = f : (B \to B) \to B \to B, x : B$.

- $(\lambda y.y)x$. El término representa la aplicación de la función identidad a la variable x.
  1. Primero tipamos el argumento x. Como $x : B \in \Gamma$, entonces a partir de la regla (T-Var) tenemos que $\Gamma \vdash x : B$. Este será el tipo de argumento (T_1) requerido en la regla (T-App).

2. Luego debemos tipar la función $(\lambda y.y)$. Para que esta función acepte el tipo B (de x), necesitamos que su tipo sea $B \to T_2$.

   Con esto dicho, aplicamos (T-Abs) con el supuesto de que $\Gamma, y : B \vdash y : B$. Y para aplicar esta regla tenemos que aplicar (T-Var), $\Gamma, y : B \vdash y : B$.

   Conclusión: $\Gamma \vdash \lambda y.y : B \to B$.

3. Finalmente, tipamos la aplicación $(\lambda y.y)x$. Aplicamos (T-App): $t_1 = (\lambda y.y)$ con tipo $B \to B$ y $t_2 = x$ con tipo B. Los tipos coinciden y luego el resultado es B.

$$\dfrac{\dfrac{\dfrac{y : B \in \Gamma, y : B}{\Gamma, y : B \vdash y : B}\ \text{(T-Var)}}{\Gamma \vdash \lambda y.y : B \to B}\ \text{(T-Abs)} \qquad \dfrac{x : B \in \Gamma}{\Gamma \vdash x : B}\ \text{(T-Var)}}{\Gamma \vdash (\lambda y.y)x : B}\ \text{(T-App)}$$

- $(\lambda g.f\ g)\ x$. No tipa la expresión (probar).

7.3.3. Recursión y normalización en lambda cálculo simplemente tipado.

El λ-cálculo simplemente tipado no permite **recursión nativa.** Términos como el operador de punto fijo $\mathbf{Y}$ del λ-cálculo puro **no pueden tiparse** en $\lambda_{\to}$:

$$\mathbf{Y} = \lambda x\ .(\lambda y\ .\ x\ (y\ y))\ (\lambda y\ .\ x\ (y\ y))$$

Podemos ver que el término $(y\ y)$ es una aplicación, entonces necesitamos que la primera y tenga tipo $y : T \to T$. Pero al ser una aplicación, también deberíamos tener que la segunda $y : T$. Es decir, $y : T \to T$ e $y : T$, contradicción. Por esto, el operador de punto fijo $\mathbf{Y}$ no puede tiparse.

Aún así, la **recursión** se puede recuperar en $\lambda_{\to}$ agregando un operador **fix,** que se agrega como constante:

```
fix : (T -> T) -> T
fix t = t (fix t)
```

Por último, a diferencia del λ-cálculo sin tipos, $\lambda_{\to}$ es **fuertemente normalizante,** dado que cualquier término es fuertemente normalizante. Es decir, toda secuencia de reducción que comienza en un término bien tipado es finita y termina en una forma normal.

7.3.4. Estrategia de evaluación para $\lambda_{\to}$.

Recordemos las estrategias de reducción:

- **Reducción estricta (Call-By-Value).** El argumento se evalúa siempre, independientemente de si se usa en el cuerpo de la función. El operador $\mathbf{Y}$ (si se añadiera) divergiría con esta estrategia: Yg diverge para cualquier g.
- **Reducción no estricta (Call-By-Name, Call-By-Need / Lazy).** El argumento se evalúa solo si es necesario (cuando se usa en el cuerpo).

Para el λ-cálculo simplemente tipado, daremos una estrategia de reglas de reducción **Call-By-Value:**

- $\dfrac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}$ (E-App1). Evalúa la función (t_1) primero.

- $\dfrac{t_2 \to t_2'}{v\ t_2 \to v\ t_2'}$ (E-App2). Si la función (v) ya es un valor, evalúa el argumento (t_2).

- $(\lambda x.t)v \to t[v/x]$ (E-AppAbs)

  Cuando ambos son valores (abstracción aplicada a un valor), realiza la β-sustitución.

Donde `v ->c |λx . t`. Las reglas (E-App1) y (E-App2) son reglas de congruencia, mientras que (E-AppAbs) es una regla de computación.

7.3.5. Seguridad formal del $\lambda_{\to}$.

El $\lambda_{\to}$ es un lenguaje seguro porque satisface las propiedades formales de progreso y preservación.

- **Progreso:** los términos bien tipados o son valores o pueden dar un paso de reducción (no se atascan).
- **Preservación:** la evaluación preserva el tipo, si $t : T$ y $t \to t'$, entonces $t' : T$.

La prueba de preservación requiere el siguiente lema de sustitución:

$$\frac{\Gamma \vdash s : S \qquad \Gamma, x : S \vdash t : T}{t[s/x] : T}$$

7.3.6. Lambda Cálculo Tipado a la Curry y a la Church.

Existen dos enfoques para la sintaxis de tipado en λ-cálculo:

- **A la Curry (implícita).** El tipo no está escrito explícitamente en la abstracción ($\lambda x.t$). El tipado se realiza por inferencia (el $\lambda_{\to}$ que hemos visto es a la Curry).
- **A la Church (explícita).** El tipo se escribe explícitamente en la abstracción ($\lambda x : T\ .\ t$).

Ambos cálculos son **equivalentes** en su poder expresivo.

7.4. Sistema T de Gödel.

7.4.1. Extensiones del sistema.

El **Sistema T** de Kurt Gödel es una extensión del λ-cálculo simplemente tipado ($\lambda_{\to}$) que recupera la capacidad de expresar todas las funciones recursivas primitivas al añadir tipos base y un operador de recursión explícito.

El **Sistema T** extiende $\lambda_{\to}$ con los siguientes elementos:

- **Tipos Base:** introduce explícitamente los tipos `Bool` y `Nat`.
- **Constantes Booleanas:** `true, false` (constructores) y `D` (eliminador *decisión*, actúa como la operación `if-then-else`).
- **Constantes Naturales:** `0, succ` (constructores) y `R` (eliminador *recursión*, que implementa la recursión primitiva para los naturales, análogo al `foldn`).

7.4.2. Reglas de tipado adicionales.

El **Sistema T** requiere reglas de tipado para sus nuevas constantes.

- $\Gamma \vdash \mathit{true} : \mathit{Bool}$ (T-True)
- $\Gamma \vdash \mathit{false} : \mathit{Bool}$ (T-False)
- $$\frac{\Gamma \vdash t_1 : \mathit{Bool} \qquad \Gamma \vdash t_2 : T \qquad \Gamma \vdash t_3 : T}{\Gamma \vdash D\ t_1\ t_2\ t_3 : T}$$ (T-D)
- $\Gamma \vdash 0 : \mathit{Nat}$ (T-0)
- $$\frac{\Gamma \vdash t : \mathit{Nat}}{\Gamma \vdash \mathit{succ}\ t : \mathit{Nat}}$$ (T-Succ)
- $$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T \to \mathit{Nat} \to T \qquad \Gamma \vdash t_3 : \mathit{Nat}}{\Gamma \vdash R\ t_1\ t_2\ t_3 : T}$$ (T-Rec)

7.4.3. Reglas de evaluación adicionales.

Estas reglas definen cómo el eliminador `D` (decisión) y `R` (recursión) computan.

- $D\ \mathit{true}\ t_2\ t_3 \to t_2$ (E-DTrue)
- $D\ \mathit{false}\ t_2\ t_3 \to t_3$ (E-DFalse)
- $$\frac{t_1 \to t_1'}{D\ t_1\ t_2\ t_3 \to D\ t_1'\ t_2\ t_3}$$ (E-D)
- $R\ t_1\ t_2\ 0 \to t_1$ (E-R0)
- $R\ t_1\ t_2\ (\mathit{succ}\ t) \to t_2\ (R\ t_1\ t_2\ t)\ t$ (E-RSucc)
- $$\frac{t_3 \to t_3'}{R\ t_1\ t_2\ t_3 \to R\ t_1\ t_2\ t_3'}$$ (E-R)

7.4.4. Programando en Sistema T: Factorial.

La potencia del Sistema T reside en el operador `R`, que permite definir funciones recursivas primitivas (aquellas cuya terminación está garantizada por la disminución del argumento natural).

Factorial en Haskell:

```
fact 0 = 1
fact (n+1) = (n+1) * fact n
```

Factorial en Sistema T:

`fact n = R 1 (λr m . prod (succ m) r) n`

- 1 es el caso base, y es el numeral de Church o constante para el uno.
- $(\lambda r\ m\ .\ prod\ (succ\ m)\ r)$ es la función que se aplica en los pasos recursivos. Recibe el resultado de la recursión r y el natural anterior m. Devuelve $r \times (m + 1)$

Simulación de evaluación. Factorial de 2.

1. Queremos calcular $fact$ 2. El término inicial es:

   `R 1 (λr m . prod (succ m) r) (succ 1)`

2. Vemos que el natural (3er argumento) es un sucesor (`succ 1`). Por lo tanto, aplicamos la regla (E-RSucc).

   `(λr m . prod (succ m) r) (R 1 (λr m . prod (succ m) r) 1) 1`

3. Evaluamos la llamada recursiva `R 1 f 1`, donde ahora el natural es `1 = succ 0`, asique aplicamos nuevamente la regla (E-RSucc).

   `(λr m . prod (succ m) r) (λr m . prod (succ m) r) ((R 1 (λr m . prod (succ m) r) 0) 0) 1`

4. Ahora aplicamos la regla (E-R0) pues tenemos que el argumento es 0.

   `(λr m . prod (succ m) r) ((λr m . prod (succ m) r) 1 0) 1`

5. Ahora aplicamos la β-reducción $r = 1$ y $m = 0$.

   `(λr m . prod (succ m) r) (prod (succ 0) 1) 1`

   `(λr m . prod (succ m) r) 1 1`

6. Aplicamos β-reducción nuevamente con $r = 1$ y $m = 0$

   `(prod (succ 1) 1) = prod 2 1 = 2`

8. Unidad 4.4 - Polimorfismo.

El **polimorfismo** es una característica fundamental de los sistemas de tipos avanzados que permite que un único fragmento de código pueda operar sobre **múltiples tipos de datos** de manera uniforme.

Antes de introducir el polimorfismo, recordemos que hemos analizado:

- $\lambda_{\rightarrow}$ **Cálculo Simplemente Tipado:** un sistema fuertemente normalizante que garantiza que todos los términos tienen una reducción finita. Sus términos son:

$$t \rightarrow x \mid c \mid t\ t \mid \lambda x.t$$

- **Sistema T de Gödel:** la extensión de $\lambda_{\rightarrow}$ con tipos de datos recursivos $(Nat, Bool)$ y operadores de recursión (R) y (D).

El siguiente paso es extender $\lambda_{\rightarrow}$ con la capacidad de definir **funciones polimórficas.**

8.1. Polimorfismo.

8.1.1. Definición. Polimorfismo.

El **polimorfismo** permite que un programa pueda ser utilizado con diferentes tipos en contextos diferentes. Existen dos categorías principales de polimorfismo.

8.1.2. Polimorfismo Ad-hoc (sobrecarga).

El **polimorfismo ad-hoc** se refiere a una función que denota **múltiples implementaciones distintas,** cada una especializada para un tipo particular, aunque compartan el mismo nombre. Es un polimorfismo por conveniencia.

Su implementación se logra mediante la **sobrecarga de operadores** o funciones. Cuando el compilador encuentra una llamada a la función, utiliza la información de los tipos de los argumentos para decidir (en tiempo de compilación) cuál de las implementaciones debe invocar.

Ejemplo. El operador de suma + que suma dos enteros es diferente al operador + que concatena dos cadenas, aunque ambos se llamen igual. En Haskell, el siguiente tipo usa la restricción `Num a` para indicar que a debe ser de una clase que soporta la operación, pero la implementación exacta del + para enteros es distinta a la del + para números en punto flotante.

```
(+) :: Num a =>a ->a ->a
```

8.1.3. Polimorfismo paramétrico.

El **polimorfismo paramétrico** permite que un único código o definición funcione sin cambios para una variedad infinita de tipos. Esto se logra introduciendo **variables de tipo** (cuantificadas universalmente $\forall$) en la signatura.

El concepto general es que la función es **paramétrica** con respecto a los tipos, es decir, el cuerpo de la función opera sobre la estructura del dato, no sobre el valor específico del tipo. El programador solo proporciona una definición para todos los casos.

Ejemplo. La función identidad recibe un valor de tipo a y devuelve exactamente el mismo valor a. No necesita saber si a es entero, una lista o un booleano para realizar su trabajo.

```
id :: a ->a
```

Variantes de polimorfismo paramétrico.

Aunque el concepto general es el mismo (el uso de variables de tipo), la forma en que el lenguaje gestiona y permite cuantificar $(\forall)$ esas variables define dos subtipos clave:

- **Polimorfismo de 1° clase.** En este polimorfismo las variables de tipo pueden cuantificarse en cualquier nivel (incluso en tipos de función). Permite pasar funciones polimórficas como argumentos. No admite inferencia de tipos general (requiere anotaciones explícitas).
- **Polimorfismo Let (ML-Style).** En este polimorfismo las variables de tipo solo pueden cuantificarse universalmente en el nivel superior de una definición. Como consecuencia no se pueden pasar funciones polimórficas como argumento. Admite inferencia de tipos.

8.2. Cálculo Lambda Polimórfico. Sistema F.

El **Sistema F** extiende el $\lambda_{\rightarrow}$ con polimorfismo paramétrico de 1° clase. La forma en que lo hace es agregando abstracciones y aplicaciones de tipo a los términos de $\lambda_{\rightarrow}$.

8.2.1. Sintaxis.

`t ::= x | c | t t | λx . t | ∧X . t | t ⟨T⟩`

`v ::= ... | ∧X . t`

`T ::= B | T → T | X | ∀X . T`

- X representa una variable de tipo, que puede ser $Nat, Bool$ o cualquier otro.
- $\wedge X.t$ es una **abstracción de tipo.** Define una función polimórfica. $\wedge$ es el operador de abstracción, X es la variable de tipo ligada, y t es el cuerpo del término.
- $t\langle T\rangle$ es una **aplicación de tipo.** Esto instancia una función polimórfica t con un tipo concreto T. Es el mecanismo para usar la función genérica.
- $\forall X.T$ es la **cuantificación universal.** Define el tipo polimórfico. Indica que el tipo T es válido para todo tipo que se sustituya por X.

8.2.2. Reglas de tipado.

Se agregan las siguientes reglas de tipado, donde el entorno Γ contiene variables de términos y de tipos:

- $$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T} \quad \text{(T-TAbs)}$$
- $$\frac{\Gamma \vdash t_1 : \forall X.T}{\Gamma \vdash t_1\langle T_2\rangle : T[T_2/X]} \quad \text{(T-TApp)}$$

La **primer regla (T-TAbs)** es el equivalente de la λ-abstracción, pero para tipos. Permite crear un término que es genérico.

El mecanismo consiste en asumir que la variable de tipo X es válida en el contexto extendido (Γ, X). Luego se verifica que el cuerpo del término t tiene tipo T. Y se concluye que el término $\wedge X.t$ (función polimórfica) tiene el tipo $\forall X.T$.

La **segunda regla (T-TApp)** permite utilizar una función polimórfica instanciándola con un tipo específico.

La premisa requiere que t_1 sea un término polimórfico con tipo $\forall X.T$. La conclusión es que el término $t_1\langle T_2\rangle$ tiene el tipo T luego de aplicar la sustitución, que significa: reemplazar cada ocurrencia de la variable de tipo X por el tipo concreto T_2 en el tipo T.

8.2.3. Reglas de semántica (evaluación).

- $\dfrac{t_1 \to t_1'}{t_1\langle T\rangle \to t_1'\langle T\rangle}$ (E-TApp)
- $(\wedge X.t)\langle T\rangle \to t[T/X]$ (E-TAppAbs)

La **segunda regla (E-TAppAbs)** es la regla de cómputo para el polimorfismo. Es el análogo de la β-reducción de valores.

Cuando un término $\wedge X.t$ (función polimórfica) recibe un tipo T (el argumento de tipo), la abstracción y la aplicación desaparecen, y el cuerpo t se evalúa con la variable de tipo X sustituida por el tipo T.

8.2.4. Ejemplos.

Los siguientes ejemplos ilustran cómo se definen funciones genéricas (polimórficas) mediante la abstracción de tipo ($\wedge X$).

- $id \equiv \wedge X.\lambda x : X.x$ y su tipo es $\forall X.X \to X$
- $double = \wedge X.\lambda f : X \to X.\lambda a : X.f(fa)$ y si tipo es $\forall X.(X \to X) \to X \to X$

8.3. Representación de tipos de datos en Sistema F.

La gran potencia del Sistema F radica en que su polimorfismo de primera clase es lo suficientemente expresivo para **tipar las codificaciones de Church** de la estructuras de datos fundamentales (booleanos, naturales, listas).

Al tipar estas estructuras, el Sistema F se vuelve totalmente autosuficiente y no requiere que existan tipos base nativos (como Nat o $Bool$), eliminando la necesidad de las constantes que se vieron en el Sistema T.

8.3.1. Representación de booleanos.

Se utilizan las representaciones de Church para booleanos, donde un booleano es una función que selecciona una de dos opciones:

$$true \equiv \lambda t\ f\ .\ t \qquad\qquad false \equiv \lambda t\ f\ .\ f$$

Introducimos el **Tipo Boleano de Church (CBool)** que cuantifica sobre el tipo de las dos ramas (X), asegurando que ambas ramas tengan un tipo común:

$$\boxed{CBool \equiv \forall X.X \to X \to X}$$

Ahora podemos definir las constantes tipadas $true$ y $false$ en Sistema F:

$$\boxed{\begin{array}{r} true \equiv \wedge X\ .\ \lambda t : X\ .\ \lambda f : X\ .\ t \\ false \equiv \wedge X\ .\ \lambda t : X\ .\ \lambda f : X\ .\ f \end{array}}$$

Luego podemos definir funciones booleanas en Sistema F:

- $not : CBool \to CBool$

  $not \equiv \lambda b : CBool\ .\ \wedge X\ .\ \lambda t : X\ .\ \lambda f : X\ .\ b\langle X\rangle\ f\ t$
- $not : CBool \to CBool$

  $not \equiv \lambda b : CBool\ .\ b\langle CBool\rangle\ false\ true$

- $and : CBool \to CBool \to CBool$

  $and \equiv \lambda b_1 : CBool \ . \ \lambda b_2 : CBool \ . \ b_1\langle CBool\rangle \ b_2 \ false$

  Podemos ver que el término b_1 se aplica al tipo $CBool$. Esto aseura que ambas ramas (b_2 y $false$) tengan el mismo tipo $CBool$. Si b_1 se reduce a $true$, seleccionará su primer argumento ($true \wedge b_2 = b_2$) . Si en cambio reduce a $false$, seleccionará su segundo argumento ($false \wedge b_2 = false$).

8.3.2. Representación de naturales.

Se utilizan los numerales de Church, donde un natural es una función que aplica una función f (el sucesor) n veces a un valor inicial x (el cero):

$$n \equiv \lambda f \ x \ . \ f^n x$$

- $\underline{0} \equiv \lambda f \ x \ . \ x$
- $\underline{1} \equiv \lambda f \ x \ . \ f \ x$
- $\underline{2} \equiv \lambda f \ x \ . \ f \ (f \ x)$

Introducimos el **Tipo Natural de Church (CNat),** que cuantifica sobre el tipo X de los elementos sobre los que opera el numeral:

$$\boxed{CNat \equiv \forall X \ . \ (X \to X) \to X \to X}$$

Ahora podemos definir los naturales en Sistema F:

- $0 : CNat$

  $0 \equiv \wedge X \ . \ \lambda f : X \to X \ . \ \lambda x : X \ . \ x$

- $suc : CNat \to CNat$

  $suc \equiv \lambda n : CNat \ . \ \wedge X \ . \ \lambda f : X \to X \ . \ \lambda x : X \ . \ f \ (n\langle X\rangle f \ x)$

8.3.3. Representación de listas.

(slide 15)

9. Unidad 5.1 - Abstracciones.

9.1. Abstracciones en programación funcional.

La **abstracción** es una técnica central en la computación que busca ocultar los detalles complejos de la implementación para revelar solo la funcionalidad esencial. En programación funcional, esto se logra a menudo a través de **clases de tipos.**

¿Por qué abstraer?

El uso estratégico de abstracciones ofrece ventajas significativas en el desarrollo de software:

- **Simplificación y legibilidad:** al encapsular lógica común o estructuras repetitivas, el código se vuelve más limpio y fácil de entender.
- **Reuso (generalización):** una abstracción exitosa actúa como una **generalización** de un concepto, permitiendo que un único conjunto de funciones (como `map`) opere sobre múltiples tipos de datos.
- **Fundamento teórico:** las mejores abstracciones se basan en teorías matemáticas sólidas (como la teoría de categorías), lo que garantiza que poseen propiedades rigurosas y predecibles.

Buenas abstracciones.

Una abstracción es considerada de alta calidad si cumple con los siguientes criterios:

- **Precisión:** posee una definición precisa y no ambigua (por ejemplo, a través de una clase de tipos y sus leyes).
- **Fundamento teórico:** está respaldada por un fundamento teórico que provee resultados y propiedades invariantes sobre su comportamiento.
- **Generalidad:** es general y útil en una amplia gama de situaciones y tipos de datos.

Ejemplos de abstracciones de categoría.

Veremos tres abstracciones fundamentales en Haskell que provienen de la **Teoría de Categorías,** organizadas jerárquicamente:

- **Functores (Functor):** generalizan el mapeo de una función.
- **Functores Aplicativos (Applicative):** generalizan la aplicación de una función.
- **Mónadas (Monad):** generalizan la noción de programación con efectos y secuencias.

9.2. Functores. Generalización del map.

9.2.1. Introducción.

La operación de aplicar una función a los elementos internos de una estructura (como `map` en listas) no es exclusiva de listas. La idea es extender esta capacidad a cualquier contenedor (por ejemplo árboles):

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

-- map de arboles
data Tree a = L a | N (Tree a) (Tree a)

mapT :: (a -> b) -> Tree a -> Tree b
```

```haskell
mapT f (L x) = L (f x)
mapT f (N l r) = N (mapT f l) (mapT f r)
```

Aún así, no todas las estructuras pueden ser functores. Consideremos la siguiente estructura que contiene una función:

```haskell
data Func a = Func (a -> a)

mapFunc :: (a -> b) -> Func a -> Func b
mapFunc g (Func h) = Func id
```

Aunque esta definición tipa (es decir, el compilador la acepta), no cumple con la lógica de mapeo, de aplicar la función que recibe sobre los elementos de la estructura, ya que ignora la función h y la función g, devolviendo una estructura que solo contiene la función identidad. Esto nos lleva a la necesidad de leyes.

9.2.2. Definición de Functor en Haskell.

En Haskell, la abstracción se define mediante la clase de tipos `Functor`:

```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Un constructor de tipos f (que toma un argumento, como [], `Maybe`, `Tree`) es un **Functor** si se puede definir `fmap` para él y satisface las siguientes dos **leyes de Functor:**

- **(Functor.1) Identidad.** Aplicar la identidad no cambia el contenedor.

  ```
  fmap id = id
  ```

- **(Functor.2) Composición.** Mapear dos funciones compuestas es equivalente a mapearlas secuencialmente.

  ```
  fmap f . fmap g = fmap (f . g)
  ```

9.2.3. Ejemplos de Functores.

```haskell
-- Functor Listas
instance Functor [] where
    fmap = map

-- Functor Maybe
data Maybe a = Nothing | Just a

instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)

-- Functor Arboles
data Tree a = L a | N (Tree a) (Tree a)

instance Functor Tree where
    fmap f (L x) = L (f x)
    fmap f (N l r) = N (fmap f l) (fmap f r)
```

Prueba de ecuaciones de functores para listas.

- Probamos **Functor.1** para listas por inducción sobre xs:

  $$\forall xs :: [a], fmap\ id\ xs = xs.$$

$$
\begin{aligned}
&= fmap\ id\ [\,] \\
&= [\,] && \text{(fmap.1)}
\end{aligned}
$$

$$
\begin{aligned}
&= fmap\ id\ (x : xs) \\
&= id\ x : fmap\ id\ xs && \text{(fmap.2)} \\
&= x : fmap\ id\ xs && \text{(id.1)} \\
&= x : xs && \text{(HI)}
\end{aligned}
$$

- Probamos **Functor.2** para listas por inducción estructural sobre la lista xs:

$$\forall xs :: [a], (fmap\ f\ .\ fmap\ g)\ xs = fmap\ (f\ .\ g)\ xs$$

Caso base [ ]

$$
\begin{aligned}
&= (fmap\ f\ .\ fmap\ g)\ [\,] \\
&= fmap\ f\ (fmap\ g\ [\,]) && \text{(def composicion)} \\
&= fmap\ f\ [\,] && \text{(fmap.1)} \\
&= [\,] && \text{(fmap.1)}
\end{aligned}
$$

$$
\begin{aligned}
&= fmap\ (f\ .\ g)\ [\,] \\
&= [\,] && \text{(fmap.1)}
\end{aligned}
$$

Caso inductivo.

$$
\begin{aligned}
&= (fmap\ f\ .\ fmap\ g)\ (x : xs) \\
&= fmap\ f\ (fmap\ g\ (x : xs)) && \text{(def. composicion)} \\
&= fmap\ f\ (g\ x : fmap\ g\ xs) && \text{(fmap.2)} \\
&= f\ g\ x :\ fmap\ f\ (fmap\ g\ xs) && \text{(fmap.2)} \\
&= (f\ .\ g)\ x : (fmap\ f\ .\ fmap\ g)\ xs && \text{(def. composicion)} \\
&= (f\ .\ g)\ x : fmap\ (f\ .\ g)\ xs && \text{(HI)} \\
&= fmap\ (f\ .\ g)\ (x : xs) && \text{(fmap.2 inversa)}
\end{aligned}
$$

Conclusión, la estructura de listas se comporta como un contenedor bien portado que permite la aplicación de una función sin alterar la estructura ni la identidad de los datos.

9.3. Functores Aplicativos. Generalización de la aplicación.

9.3.1. Introducción.

Los **Functores Aplicativos** son una extensión de los functores que permiten generalizar el mapeo de funciones con cualquier número de argumentos.

Si solo usamos `fmap`, podemos aplicar una función solo si ya tenemos todos los argumentos envueltos en la estructura de f:

```
-- Map aplicado a dos argumentos.
fmap2 :: (a -> b -> c) -> f a -> f b -> f c

-- Map aplicado a tres argumentos.
fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
```

Para evitar definir estas funciones infinitamente, introducimos dos operaciones base que permiten la aplicación secuencial de argumentos envueltos (currificados).

9.3.2. Definición de Functor Aplicativo en Haskell.

La clase `Applicative` requiere la existencia de un Functor y se define de la siguiente manera en Haskell:

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

- **pure (Envoltura).** Esta función tiene como propósito introducir un valor simple a al contexto f (lo envuelve) sin añadir efectos o contexto. Ejemplo, en $Maybe$, `pure 5` devuelve `Just 5`. En listas, `pure 5` devuelve `[5]`.
- $< * >$ **(Aplicación Envuelve).** Es el operador de aplicación envuelta: toma una función ya envuelta (`f (a ->b)`) y le aplica un argumento envuelto `(f a)` para producir un resultado envuelto `(f b)`.

La verdadera potencia del Applicative es que permite aplicar funciones de N argumentos simplemente encadenando $< * >$. Esto se logra gracias a que `fmap` inicia la secuencia al currificarse.

El concepto de functor aplicativo también viene de la teoría de categorías. **El buen comportamiento de un functor aplicativo queda especificado por las siguientes leyes:**

- **Identidad:** `pure id <*> x = x`
- **Composición:** `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- **Homomorfismo:** `pure f <*> pure x = pure (f x)`
- **Intercambio:** `u <*> pure y = pure (\g ->g y) <*> u`

9.3.3. Ejemplos de Functores Aplicativos.

```
instance Functor Maybe => Applicative Maybe where
    pure = Just
    (Just f) <*> (Just x) = Just (f x)
    _ <*> _ = Nothing
```

Ejemplos aplicando el Functor Aplicativo Maybe:

```
pure (+2) <*> Just 4
-- Just (+2) <*> Just 4 -> Just (+2 4) -> Just 6

pure (+3) <*> Just 7
-- Just (+3) <*> Just 7 -> Just (+3 7) -> Just 10

pure (+) <*> Just 3 <*> Just 5
-- Just (+) <*> Just 3 <*> Just 5 -> Just (+3) <*> Just 5 -> Just (+3 5) -> Just 8

pure (+) <*> Nothing <*> Just 2
-- Just (+) <*> Nothing <*> Just 2 -> Nothing
```

```
instance Functor [] => Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

Ejemplos aplicando el Functor Aplicativo de listas:

```
[(+1), (^2)] <*> [1,2,3]
--  [2,3,4,1,4,9]
```

```
[(+), (^)] <*> [3,5] <*> [2,3]
-- [(+3), (+5), (3^), (5^)] <*> [2,3] -> [5,6,7,8,9,27,25,125]
```

9.4. Mónadas. Abstracción para computaciones con efectos.

9.4.1. Introducción.

El concepto de **Mónada** proviene de la Teoría de Categorías y fue introducido en la semántica de los lenguajes de programación por Eugenio Moggi (1991). Posteriormente, Philip Wadler y Simon Peyton Jones aplicaron estos principios para dar estructura a los programas funcionales de Haskell, resolviendo el desafío de manejar **funciones impuras** y **efectos secundarios** en un entorno de funciones puras.

Funciones Puras vs. Impuras.

Para entender la necesidad de las Mónadas, es fundamental diferenciar los tipos de funciones en un contexto funcional:

- **Función Pura.** Una función es **pura** si su resultado depende únicamente de sus argumentos y no produce efectos secundarios (modificaciones al estado externo o I/O). Su evaluación siempre devuelve el mismo resultado para las mismas entradas.
- **Función Impura.** El resultado puede depender de entradas externas (por ejemplo, lectura de teclado) o puede modificar el estado externo (por ejemplo, escritura a un archivo, impresión en pantalla).

Las Mónadas proporcionan una forma de **encapsular los efectos impuros** dentro de una estructura de datos pura, permitiendo manejarlos de manera controlada y secuencial.

9.4.2. Mónadas como extensión de Functores Aplicativos.

McBride y Paterson mostraron que los functores aplicativos generalizan a las mónadas. Toda mónada es un functor aplicativo.

$$\text{Mónada} \Rightarrow \text{Functores aplicativos} \Rightarrow \text{Functores}$$

Las mónadas extienden la funcionalidad de los functores aplicativos. Mientras que un functor aplicativo f permite combinar valores envueltos de manera independiente (por ejemplo, `f a <*> f b`), **la mónada permite secuenciar computaciones donde el resultado de la primera determina la siguiente acción.**

El problema de la dependencia secuencial.

Supongamos que tenemos un valor envuelto de tipo `m a` (donde `m` es el contexto monádico, por ejemplo `Maybe`). Queremos aplicar una función k a su contenido a, pero esta función k también devuelve un resultado envuelto: `k :: a ->m b`.

El functor aplicativo no puede hacer esto porque terminaría con un tipo anidado: `m (m b)`. Se necesita una función que *aplane* o *una* el resultado:

```
f a -> (a -> f b) -> f b
```

9.4.3. Definición de la Clase Monad en Haskell.

La clase `Monad` se define sobre la clase `Applicative` y requiere dos funciones clave que resuelven este aplanamiento y secunciación:

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

- **return.** Es la versión monádica de `pure`. Envuelve un valor simple a en el contexto monádico m, siendo la forma más pura de crear un valor $m\ a$.
- **($\gg=$) (Bind o Enlace).** Es el corazón de la mónada. Toma un valor envuelto ($m\ a$) y una función que produce un valor envuelto ($a \to m\ b$), ejecutando la secuencia y **aplanando el doble contexto** para devolver un simple $m\ b$.

Para que un constructor de tipos m sea una mónada válida y bien comportada, debe satisfacer las siguientes tres leyes:

- **Monad.1:** `return a >>= k = k a`
- **Monad.2:** `m >>= return = m`
- **Monad.3:** `m >>= (\x -> k x >>= h)`

9.4.4. Ejemplos de Instancias Monádicas.

- **Instancia Lista [].** La mónada lista modela la computación no determinista o la generación de múltiples resultados posibles.

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
```

  - `return`. Envuelve un valor en una lista unitaria.
  - $\gg=$. Aplica la función `f :: a ->[b]` (tipo requerido por el operador `bind`) a cada elemento de la lista xs, generando una lista de listas, y luego usa `concat` para aplanar el resultado.

  Ejemplo de uso:

```
> [1,2,3] >>= (\x -> [x,-x])
concat (map (\x -> [x,-x]) [1,2,3]) -> concat [[1,-1], [2,-2], [3,-3]] -> [1,-1,2,-2,3,-3]
```

- **Instancia Maybe.** La mónada Maybe modela programas que pueden fallar.

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f = f x
```

  - `return`. Envuelve un valor en `Just`.
  - $\gg=$. Si el valor inicial es `Nothing` el enlace se detiene y propaga `Nothing`. Si es `Just x`, el contenido de x se extrae, se aplica la función f, y el resultado envuelto es devuelto.

  Ejemplo de uso:

```
> Just 5 >>= \x -> return (x*10)
-- (\x -> return (x*10)) (Just 5) -> Just 50
```

```
> Nothing >>= \x -> return (x*10)
-- Nothing
```

Se puede probar fácilmente que se cumplen las 3 leyes monádicas para ambas instancias (listas y Maybe).

9.4.5. Resumen.

- Vimos 3 abstracciones más que vienen de teoría de categorías: functores, functores aplicativos y mónadas. En Haskell, estas abstracciones se definen mediante clases y el programador debe chequear que se satisfacen las leyes de buen comportamiento.
- Vimos que los functores aplicativos y las mónadas pueden usarse para combinar computaciones no deterministas y computaciones que pueden fallar.
- Con los functores aplicativos se utiliza un estilo de programación aplicativo, mientras que con las mónadas (con la notación *do*) un estilo secuencial (o imperativo).
- Se puede probar que todo mónada es un functor, proveyendo una instancia y probando que se cumplen las leyes de functores:

```
instance Monad m => Functor m where
    fmap f m = m >>= (\x -> return (f x))
```

- También se puede probar que todo mónada es un functor aplicativo, proveyendo las instancias `pure` y `<*>`:

```
instance Monad m => Applicative m where
    pure x = return x
    f <*> x = f >>= (\g -> x >>= (\y -> return (g y)))
```

10. Unidad 5.2 - Modelando efectos computacionales con mónadas.

10.1. Introducción.

10.1.1. Repaso.

La clase `Monad` extiende la clase `Applicative` y define la capacidad de secuenciar computaciones dependientes.

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

Toda instancia monádica debe satisfacer las siguientes leyes monádicas que garantizan la coherencia de la secuenciación:

- **Monad.1:** `return a >>= k = k a`
- **Monad.2:** `m >>= return = m`
- **Monad.3:** `m >>= (\x -> k x >>= h)`

Hemos visto que la mónada `Maybe` modela el manejo de fallos y `[]` el no determinismo.

10.1.2. Objetivos.

El objetivo es demostrar que las mónadas son un patrón de diseño universalmente aplicable para transformar código que maneja efectos (imperativo o con lógica de control compleja) en un código limpio, legible y estructurado. Modelaremos:

- Manejo de errores (usando `Maybe`).
- Manejo de estado o log (usando `Writer` o `Acum`).
- Lectura de valores asociados a variables en un entorno.

10.2. Caso 1. Manejo de errores con mónada Maybe.

Usaremos la mónada `Maybe` para refactorizar un evaluador de expresiones aritméticas (`Exp`) y manejar el error de división por cero.

10.2.1. Evaluador puro (sin manejo de errores, sin mónadas).

El evaluador puro no maneja la división por cero y simplemente usa la división entera `div`:

```
data Exp = Lit Int | Add Exp Exp | Div Exp Exp

eval :: Exp -> Int
eval (Lit n) = n
eval (Add t u) = eval t + eval u
eval (Div t u) = div (eval t) (eval u)
```

10.2.2. Evaluador con manejo de error (sin mónadas).

Para manejar el error, la versión manual (sin mónadas) se vuelve muy verbosa, requiriendo `case` anidados para chequear la propagación del fallo en cada paso:

```haskell
eval :: Exp -> Maybe Int
eval (Lit n) = Just n
eval (Add t u) = case (eval t) of
    Nothing -> Nothing
    Just x -> case (eval u) of
        Nothing -> Nothing
        Just y -> Just (x+y)
eval (Div t u) = case (eval t) of
    Nothing -> Nothing
    Just x -> case (eval u) of
        Nothing -> Nothing
        Just y -> if y == 0
            then Nothing
            else Just (div x y)
```

10.2.3. Evaluador imperativo monádico con manejo de errores.

El uso de `bind` y `return` permiten que la mónada se encargue de la lógica de propagación de `Nothing` automáticamente.

```haskell
throw :: Maybe a
throw = Nothing

eval :: Exp -> Maybe Int
eval (Lit n) = return n
eval (Add t u) =
    eval t >>= \x ->
    eval u >>= \y ->
    return (x + y)
eval (Div t u) =
    eval t >>= \x ->
    eval u >>= \y ->
    if y == 0
        then throw
        else return (div x y)
```

La secuencia $\gg=$ solo se ejecuta si la evaluación de `eval t` y `eval u` son exitosas (`Just`). Si alguno devuelve `Nothing`, el bind lo propaga inmediatamente, sin ejecutar el resto del cuerpo.

10.2.4. Notación do.

En general un programa monádico tiene la siguiente estructura:

```haskell
m1 >>= \x1 ->
m2 >>= \x2 ->
...
return (f x1 x2 ... xn)
```

Haskell provee una sintaxis especial que mejora la legibilidad:

```haskell
do  x1 <- m1
    x2 <- m2
    ...
    return (f x1 x2 ... xn)
```

10.2.5. Evaluador imperativo monádico con notación do.

```
eval :: Exp -> Maybe Int
eval (Lit n) = return n
eval (Add t u) =
    do  x <- eval t
        y <- eval u
        return (x+y)
eval (Div t u) =
    do  x <- eval t
        y <- eval u
        if y == 0
            then throw
            else return (div x y)
```

10.3. Caso 2. Manejo de estado (log) con Acum y Writer.

El siguiente efecto a modelar es la capacidad de **acumular información** (un log, un contador, o estado) a lo largo de una computación, además de devolver un resultado.

10.3.1. Evaluador que cuenta operaciones (estilo manual).

Este evaluador devuelve una tupla (reusltado, contador), donde la segunda componente se suma en cada paso, simulando contar la cantidad de operaciones realizadas:

```
eval2 :: Exp -> (Int, Int)
eval2 (Lit n) = (n, 0)
eval2 (Add t u) =
    let (m, cm) = eval2 t
        (n, cn) = eval2 u
    in (n + m, cm + cn + 1)
eval (Div t u) =
    let (m, cm) = eval2 t
        (n, cn) = eval2 u
    in (div n m, cm + cn + 1)
```

El problema es que la gestión de la tupla (desempaquetar, combinar, reempaquetar) es manual y dispersa la lógica.

10.3.2. Mónada Acum.

Definimos una `newtype` para encapsular el efecto de acumulación y definiremos la mónada `Acum` que encapsula la lógica de combinación de estados:

```
newtype Acum a = Ac {runAc :: (a, Int)}

instance Monad Acum where
    return x = Ac (x, 0)
    Ac (x, n) >>= f =
        let Ac (x', n') = f x
        in Ac (x', n + n')
```

- `newtype Acum a`. Declaración del nuevo tipo. `Acum` es el constructor de tipos y `a` es el tipo del resultado de la computación. El efecto es el `Int`.
- `Ac`. Es el constructor de datos, es la etiqueta que usamos para crear un valor `Acum` y para desempaquetarlo.

- `runAc :: (a, Int)`. Definición de un campo record, es el extractor o funcion de acceso. Por ejemplo `runAc (Ac (5,0)) = (5,0)`
- `return x = Ac (x, 0)`. cuando se envuelve un valor `x` usando `return`, asumimos que esta es la computación más básica y no ha consumido ninguna operación. Por eso, el contador asociado es 0.
- $\gg=$. El contexto anterior `Ac (x,n)` se desestructura, x es el resultado de la computación anterior y n el contador acumulado hasta ahora.

  Luego, se aplica la función f al resultado anterior x. La función f devuelve una nueva computación monádica `Ac (x', n')`, donde x' es el nuevo resultado y n' el nuevo contador generado por la computación $f\ x$.

  Finalmente, se construye el nuevo `Acum` que devuelve el resultado (x') y el contador final $(n + n')$.

10.3.3. Evaluador que cuenta operaciones usando Acum.

La lógica de sumar los contadores ha sido abstraída dentro de la definición del `Monad Acum`.

```
-- Computacion que no devuelve valor pero agrega 1 al log.
tick :: Acum ()
tick = Ac ((), 1)

eval2 :: Exp -> Acum Int
eval2 (Lit n) = return n
eval2 (Add t u) =
    do  x <- eval2 t
        y <- eval2 u
        tick
        return (x+y)
eval (Div t u) =
    do  x <- eval2 t
        y <- eval2 u
        tick
        return (div x y)
```

10.3.4. Mónada Writer (generalización de Acum).

La mónada `Writer` es la abstracción utilizada para modelar computaciones que, además de devolver un resultado, generan un log o acumulan un valor secundario.

Es una generalización de `Acum`. Mientras que `Acum` devuelve un `Int` en la segunda componente de la tupla, `Writer` permite que sea cualquier tipo `w` siempre y cuando sea un **monoide.**

```
newtype Writer w a = Writer {runW :: (a,w)}

class Monoid a where
    mempty :: a
    mappend :: a -> a -> a

instance (Monoid w) => Monad (Writer w) where
    return x = Writer (x, mempty)
    (Writer (x,v)) >>= f =
        let (Writer (y,v')) = f x
        in Writer (y, mappend v v')
```

- `mempty`. Es el elemento neutro del monoide. Es utilizado por `return` para comenzar el log con un valor inicial nulo.

- `mappend`. Es la operación binaria de combinación, utilizada por $\gg=$ para unir el log antiguo con el nuevo log.
- Ejemplo: si `a = Int`, `mempty = 0`, `mappend = (+)` (suma).
- Ejemplo: si `a = String`, `mempty = ""`, `mappend = (++)` (concatenación).

10.4. Caso 3. Lectura de entorno con mónada Reader.

10.4.1. Evaluador con lectura de entorno.

Extendemos el lenguaje de expresiones con variables con nombres de tipo `String`.

```
    data Exp = ... | Var String

    type Env = String -> Int

    eval3 :: Exp -> Env -> Int
    eval3 (Lit n) e = n
    eval3 (Var v) e = e v
    eval3 (Plus t u) e = eval3 t e + eval3 u e
    eval3 (Div t u) e = div (eval3 t e) (eval3 u e)
```

Podemos ver que el entorno `Env` debe pasarse explícitamente a cada función recursiva, incluso a funciones que no lo necesitan.

10.4.2. Mónada Reader.

La mónada **Reader** encapsula una función que toma el entorno y devuelve el resultado.

```
newtype Reader a = Reader { runR :: (Env -> a) }

instance Monad Reader where
    return x = Reader (\_ -> x)
    Reader h >>= f = Reader (\e -> runR (f (h e)) e)

ask :: Reader Env
ask = Reader id
```

- `return`. Crea una función que ignora cualquier entorno que reciba y simplemente devuelve el valor x.
- $\gg=$. Definimos una nueva función que toma el entorno e, calcula $h\ e$ (resultado de la primera computación) y aplica la función monádica f al resultado.
- `ask`. Es una computación que, cuando se le da el entorno e, simplemente devuelve e como su resultado.

10.4.3. Evaluador monádico con Reader.

La propagación de `Env` ahora es implícita, mejorando la legibilidad.

```
eval3 :: Exp -> Reader Int
eval3 (Lit n) = return n
eval3 (Var v) =
    do  e <- ask
        return (e v)
eval3 (Plus t u) =
    do  x <- eval3 t
        y <- eval3 u
```

```
        return (x+y)
eval3 (Div t u) =
    do  x <- eval3 t
        y <- eval3 u
        return (div x y)
```

10.5. Caso 4. Combinar efectos con la mónada compuesta M.

El desafío final es combinar todos los efectos estudiados: `Reader` para entorno, `Maybe` para error, `Acum` para contador, en una única estructura `M`.

```
newtype M a = M { runM :: Env -> Maybe (a, Int) }

instance Monad M where
    return x = M (\_ -> Just (x, 0))
    M h >>= f = M (\e -> case h e of
        Nothing -> Nothing -- Manejo de error del lado izquierdo (h)
        Just (a, m) -> case runM (f a) e of -- Pasa el entorno
            Nothing -> Nothing -- Manejo de error del lado derecho (f a)
            Just (b, n) -> Just (b, m+n)) -- Exito: combina acumuladores

-- Lanzar error
throw :: M a
throw = M (\_ -> Nothing)

-- Obtiene entorno
ask :: M Env
ask = M (\e -> Just (e, 0))

-- Acumula 1
tick :: M ()
tick = M (\_ -> Just ((), 1))
```

- La estructura de `runM` se lee de afuera hacia adentro. `Env` es el efecto `Reader`, `Maybe` es el efecto de error y `(a, Int)` es el efecto `Acum`, la computación devuelve resultado `a` y contador `Int`.
- `M` se comporta como una mónada `Maybe` por fuera (si falla, devuelve `Nothing`) y como `Acum` por dentro (si tiene éxito, suma los contadores m y n).

Definimos el evaluador monádico con la mónada compuesta `M`.

```
eval :: Exp -> M Int
eval (Lit n) = return n
eval (Var v) =
    do  e <- ask
        return (e v)
eval (Plus t u) =
    do  x <- eval t
        y <- eval u
        tick
        return (x+y)
eval (Div t u) =
    do  x <- eval t
        y <- eval u
        if y == 0
            then throw
            else return (div x y)
```

10.6. Conclusiones finales.

10.6.1. Observaciones.

- Cada evaluador tiene una estructura similar, la cual pudo abstraerse usando la noción de mónada.
- En cada evaluador se introdujo un tipo de cómputo, donde el constructor monádico M representó cómputos: que pueden fallar, que llevan un acumulador, que leen de un entorno.
- Las funciones de tipo `a ->b` se reemplazaron por funciones de tipo `a ->M b`, las cuales toman un valor de tipo `a` y devuelve un valor de tipo `b` con un posible efecto adicional capturado por `M`.
- Una mónada es una abstracción. Para que la abstracción esté bien usada las funciones que emplea la mónada sólo deben utilizar la interfaz del sistema: `return`, $\gg=$ y operaciones propias de la mónada (`throw, tick, ask`).
- Las mónadas permiten escribir código más modular y reusable pero también pueden utilizarse otras estructuras, tal vez menos intuitivas que las mónadas, como arrows o functores aplicativos.

10.6.2. Operaciones de las mónadas.

Las operaciones soportadas por cada mónada pueden definirse en una clase. Por ejemplo:

```
1  class Monad m => MonadThrow m where
2      throw :: m a
3
4  class Monad m => MonadAcum m where
5      tick :: m ()
6
7  class Monad m => MonadReader m where
8      ask :: m Env
9
10 eval :: (MonadThrow m, MonadAcum m, MonadReader m) => Exp -> m Int
```