



**Universidad Nacional Autónoma de México**

**Facultad de Ciencias  
Reporte de Actividad Docente**

**MANUAL DE EJERCICIOS PARA LA MATERIA DE LENGUAJES DE  
PROGRAMACIÓN.**

**QUE PARA OBTENER EL TÍUTLO DE:**

Licenciado en Ciencias de la Computación.

**PRESENTA: LUIS MIGUEL MUÑOZ BARÓN**

**TUTOR: JAVIER ENRIQUEZ MENDOZA**

2023



## **Nota al lector**

El presente trabajo es una compilación de ejercicios teóricos para la materia de lenguajes de programación de la carrera en ciencias de la computación siguiendo el plan de estudios impartido desde el año 2013 con clave 1536 en la Facultad de Ciencias de la Universidad Nacional Autónoma de México.

El objetivo principal de este material es que sirva como guía para desarrollar los contenidos visitados en este curso siendo un apoyo didáctico para profesores y alumnos donde cada ejercicio planteado durante el capítulo esté acompañado de la base teórica para poder seguir el desarrollo del mismo y ser entendido en su totalidad.

Cada capítulo cuenta con dos secciones: una sección de teoría autocontenido con ejercicios resueltos y una sección de ejercicios sin respuesta para el lector al final del mismo.

Las demostraciones y desarrollo de los contenidos teóricos serán discutidos de forma laxa dejando a discreción del lector el estudio a profundidad de la información aquí presentada concentrándonos únicamente en la teoría necesaria para poder dar solución a los ejercicios, asimilar los conceptos y definiciones, familiarizarnos con la explicación, planteamiento y respuesta del material aquí presentado.

Para un estudio en profundidad se refiere a las personas interesadas a consultar [1], [2], [5], [6], [7], [8], [9], [10]. [12], [13], [21], [22], [24], [26], [27], [30], [32], [33], [34] de la bibliografía que se encuentra al final del presente manual.

Finalmente espero que las personas que hagan uso del presente material encuentren útil y fructífero los desarrollos y conceptos escritos en las páginas de este texto.

Miguel Barón.



*"A language is not just words. It's a culture, a tradition, a unification of a community, a whole history that creates what a community is. It's all embodied in a language"*

*-Noam Chomsky*



# Índice general

<b>1. Introducción</b>	<b>1</b>
1. Reglas de los lenguajes . . . . .	2
2. Gramáticas y semántica de expresiones . . . . .	3
3. Clasificación de los lenguajes de programación . . . . .	6
4. Ejercicios para el lector . . . . .	9
<b>2. Herramientas matemáticas</b>	<b>10</b>
1. Juicios lógicos . . . . .	11
2. Definiciones inductivas, reglas de inferencia e inducción estructural . . . . .	12
3. Sistemas de transición . . . . .	17
4. Ejercicios para el lector . . . . .	19
<b>3. Sintáxis</b>	<b>23</b>
1. Sintáxis concreta . . . . .	25
2. Sintáxis abstracta . . . . .	26
2.1. Analizador sintáctico . . . . .	26
3. El operador let . . . . .	28
3.1. Clasificación de variables en el operador let . . . . .	29
4. Sustitución y $\alpha$ -equivalencias . . . . .	32
5. Ejercicios para el lector . . . . .	35

<b>4. Semántica</b>	<b>37</b>
1. Semántica estática . . . . .	38
2. Semántica dinámica . . . . .	40
2.1. Semántica operacional . . . . .	40
2.2. Semántica de paso pequeño . . . . .	41
2.3. Semántica de paso grande . . . . .	44
3. La función eval . . . . .	46
4. Ejercicios para el lector . . . . .	47
<b>5. Cálculo Lambda</b>	<b>50</b>
1. Sintáxis del Cálculo Lambda . . . . .	52
2. $\alpha$ -equivalencia en el Cálculo Lambda . . . . .	53
3. Semántica operacional del Cálculo Lambda . . . . .	53
4. Definibilidad Lambda . . . . .	54
4.1. Booleanos y operadores lógicos . . . . .	54
5. Aritmética del Cálculo Lambda . . . . .	55
5.1. Numerales de Church . . . . .	56
5.2. Funciones aritméticas . . . . .	56
6. Datos estructurados en el Cálculo Lambda . . . . .	60
6.1. Tuplas . . . . .	60
6.2. Listas . . . . .	60
7. Propiedades semánticas del Cálculo Lambda . . . . .	61
7.1. No terminación . . . . .	61
7.2. No determinismo . . . . .	62
7.3. Confluencia . . . . .	62
8. Combinadores de punto fijo . . . . .	62

9.	Ejercicios para el lector . . . . .	64
<b>6.</b>	<b>MinHs</b>	<b>66</b>
1.	Sintáxis de MinHaskell . . . . .	67
1.1.	Sintáxis concreta . . . . .	67
1.2.	Sintáxis abstracta . . . . .	69
2.	Semántica de MinHaskell . . . . .	70
2.1.	Sistemas de tipos . . . . .	70
2.2.	Semántica estática . . . . .	70
2.3.	Semántica dinámica . . . . .	73
3.	Propiedades de MinHaskell . . . . .	74
3.1.	Seguridad del lenguaje . . . . .	74
3.2.	No terminación . . . . .	75
4.	Ejercicios para el lector . . . . .	76
<b>7.</b>	<b>Inferencia de tipos</b>	<b>78</b>
1.	Estandarización de variables . . . . .	79
2.	Generación de restricciones . . . . .	81
3.	Algoritmo de unificación . . . . .	83
4.	Algoritmo de inferencia de tipos . . . . .	83
5.	Ejercicios para el lector . . . . .	99
<b>8.</b>	<b>Máquinas abstractas</b>	<b>100</b>
1.	La Máquina $\mathcal{H}$ . . . . .	101
1.1.	Marcos y la pila de control . . . . .	102
1.2.	Estados . . . . .	102
1.3.	Transiciones . . . . .	103

2.	La Máquina $\mathcal{J}$	106
2.1.	Marcos	107
2.2.	Ambientes	107
2.3.	Estados	107
2.4.	Transiciones	108
2.5.	Alcance	110
2.6.	Closures	111
3.	Ejercicios para el lector	114
<b>9.</b>	<b>TinyC</b>	<b>115</b>
1.	Sintáxis	116
2.	Semántica operacional	118
2.1.	La Máquina C	118
2.2.	Marcos	118
2.3.	Estados	119
2.4.	Transiciones	119
3.	Ejercicios para el lector	125
<b>10.</b>	<b>Herencia y subtipos</b>	<b>126</b>
1.	Orientación a objetos	127
2.	Subtipos	128
2.1.	Subtipado de tipos primitivos	129
2.2.	Subtipado de funciones	130
2.3.	Subtipado para suma y producto	130
2.4.	Subtipado para registros	130
2.5.	Elementos máximos para el sistema de tipos	131
3.	Casting	131

3.1. <i>Upcasting</i> . . . . .	132
3.2. <i>Downcasting</i> . . . . .	132
4.    Ejercicios para el lector . . . . .	133
<b>11. Java Peso Pluma</b>	<b>134</b>
1.    Sintáxis de Java Peso Pluma . . . . .	135
2.    Tablas de clases . . . . .	138
3.    Semántica dinámica . . . . .	139
4.    Semántica estática . . . . .	140
5.    Propiedades de Java Peso Pluma . . . . .	141
5.1.    Preservación de tipos . . . . .	141
5.2.    Progreso . . . . .	141
5.3.    Seguridad . . . . .	142
6.    Cómo se relaciona Java Peso Pluma con Java? . . . . .	142
7.    Ejercicios para el lector . . . . .	144



# Capítulo 1

## Introducción



En el contexto de la computación, los lenguajes de programación fungen como una interfaz entre programador y procesador para establecer una comunicación sobre lo que queremos que la computadora ejecute por nosotros.

Diferentes paradigmas de lenguajes de programación, estilos, reglas y convenciones se han adoptado en las últimas décadas para ajustarse al mejor planteamiento posible de un problema, mantener un estándar declarativo, un nombramiento homogéneo o un estilo idéntico compartido entre los diferentes programas que se agrupan en estas categorías.

Iniciaremos el estudio de este manual planteando los conceptos claves que nos permitirán definir, clasificar y analizar a los lenguajes de programación que existen en el contexto de las ciencias de la computación. Éstos conceptos nos permitirán definir nuestro propio lenguaje de programación desde cero, creando las estructuras y reglas necesarias para cada tipo, variable e instrucción del mismo.

Por último analizaremos sus propiedades y estudiaremos sus características más importantes para poder dar una implementación robusta aplicando las definiciones que iremos presentando en el desarrollo de esta introducción.

### Objetivo

## Subsección

Comenzar el estudio formal de los lenguajes de programación brindando un primer acercamiento a la construcción básica de uno, partiendo de las definiciones de conceptos que componen un lenguaje, específicamente: sintáxis, semántica, y pragmática. Así como a las clasificaciones que agrupan y distinguen los diferentes paradigmas que existen para los lenguajes computacionales<sup>1</sup>.

### Planteamiento

Este capítulo inicia con el estudio de los lenguajes de programación definiendo EAL: Expresiones aritmético-lógicas dando la definición de la gramática y esbozando como podemos interpretar las expresiones que pertenecen al mismo mediante la función de evaluación `eval`. Este lenguaje será utilizado en los siguientes capítulos del manual para ilustrar propiedades, conceptos y definiciones mediante la extensión de las instrucciones y niveles para la interpretación del mismo.

Finalmente se concluye discutiendo los diferentes paradigmas en los que la computación moderna agrupa los lenguajes de programación en sus diferentes categorías.

## 1 Reglas de los lenguajes

La lingüística tiene como propósito definir una serie de reglas que ríjan la estructura fundamental que compone un lenguaje. Las reglas que estudian dicha composición pueden ser clasificadas en sintácticas (estructura grammatical), semánticas (significado) y pragmáticas (contexto de uso).

---

<sup>1</sup>Para las personas que poseen una madurez matemática superior a aquella que la que un estudiante de cuarto semestre de la Facultad de Ciencias pudiera tener se recomienda encarecidamente iniciar el estudio de este manual en el capítulo 3: Sintáxis. Los temas recomendados para omitir los primeros dos capítulos del manual son: estructuras discretas, juicios lógicos, reglas inductivas, reglas de inferencia e inducción estructural.

**Ejercicio 1.1.** Explica que es la sintaxis de un lenguaje.

"En el contexto de las ciencias de la computación, la sintaxis de un lenguaje son las reglas que definen las combinaciones de símbolos que se consideran declaraciones o expresiones correctamente estructuradas. Ésto se aplica tanto a los lenguajes de programación, donde el documento representa el código fuente, como a los lenguajes de marcado, donde el documento representa datos"<sup>a</sup>.

<sup>a</sup>Esta definición fue extraída del elemento [29] perteneciente a la bilbiografía de este trabajo, para un estudio en profundidad de este tema se refiere al lector a los siguientes trabajos: [30], [31], [32], [33], [34], [35]

**Ejercicio 1.2.** Explica qué es la semántica de un lenguaje.

"En el contexto de las ciencias de la computación, la semántica describe los procesos que sigue una computadora al ejecutar un programa en un lenguaje específico. Ésto se puede mostrar describiendo la relación entre la entrada y la salida de un programa, o una explicación de cómo se ejecutará el programa en una determinada plataforma, creando así un modelo de cálculo"<sup>a</sup>.

<sup>a</sup>Este dafinición fue extraída del siguiente elemento de la bibliografía: [28] y se refiere al lector a los siguientes publicaciones para un estudio en profundidad del tema: [20], [21], [22], [23], [24], [25], [26] y [27]

**Ejercicio 1.3.** Explica qué es la pragmática de un lenguaje.

"En lingüística y campos relacionados, la pragmática es el estudio de cómo el contexto contribuye al significado"<sup>a</sup>.

Particularmente en ciencias de la computación la prágmatica se refiere a todos aquellos elementos del lenguaje que nos permiten obtener el significado de una expresión bajo un determinado contexto. Un ejemplo de ésto son las bibliotecas que nos permiten importar métodos previamente definidos, así la interpretación de una instrucción de esta biblioteca solo tiene un significado válido bajo el contexto que la importa<sup>b</sup>.

<sup>a</sup>Definición extraída de [36] y [37]

<sup>b</sup>Extraido de [1], [5] y [12]

## 2 Gramáticas y semántica de expresiones

Los lenguajes necesitan de una estructura que nos permita formar expresiones pertenecientes a ellos y darles significado. Es por eso que se requiere de un conjunto de reglas que conforman la columna vertebral del lenguaje y una función que nos diga que significan aquellos elementos formados por el conjunto de reglas.

En cursos como autómatas y lenguajes formales<sup>2</sup> se revisan este tipo de estructuras y las reglas que las generan conocidas como sintaxis del lenguaje y que pueden ser definidas mediante juicios lógicos ó gramáticas.

Adicionalmente se necesita definir una función de evaluación que nos permita obtener el valor asociado a la expresión. A esta rama se le conoce como semántica del lenguaje<sup>3</sup>.

**Ejercicio 2.1.** Define la gramática para las cadenas de la forma  $(ab)^*$ .

$S ::= ab \ S \mid \epsilon$

**Ejercicio 2.2.** Define la gramática para números enteros.

$S ::= digit \mid digit \ S$   
 $digit ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

**Ejercicio 2.3.** Define la gramática para las listas de dígitos.

$S ::= [ \ digit - list \ ] \mid [ \epsilon \ ]$   
 $digit - list ::= digit \mid digit \ , \ digit - list$   
 $digit ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

**Definición 2.1** (Sintaxis de Expresiones aritmético-lógicas: EAL).

$e ::= n \mid \text{True} \mid \text{False} \mid e + e \mid e * e \mid e < e \mid \text{iszzero } e \mid \text{not } e$

En donde  $n \in \mathbb{N}$  es un natural y las constantes  $\text{False}$ ,  $\text{True} \in \text{Bool}$ <sup>a</sup>.

---

<sup>a</sup>Definición formulada de [1], [5], [12], [38] y [39].

**Definición 2.2** (Semántica de EAL). Función de evaluación para el lenguaje de expresiones aritmético-lógicas.

$$[\cdot] : \text{EAL} \rightarrow \mathbb{N} \cup \text{Bool}$$

Ésto quiere decir que la función semántica recibe una expresión del lenguaje EAL y regresa un número natural ó una constante booleana y se define de la siguiente forma:

---

<sup>2</sup>Conforme al plan de estudios que se imparte desde el 2013 en la Facultad de Ciencias de la Universidad Nacional Autónoma de México con clave de asignatura 1425.

<sup>3</sup>Más adelante en este curso revisaremos a profundidad las reglas semánticas y como evaluar expresiones correctas.

$$\begin{aligned}
\llbracket n \rrbracket &= n \\
\llbracket \text{True} \rrbracket &= \text{True} \\
\llbracket \text{False} \rrbracket &= \text{False} \\
\llbracket e_1 + e_2 \rrbracket &= \llbracket e_1 \rrbracket +_{\mathbb{N}} \llbracket e_2 \rrbracket \\
\llbracket e_1 * e_2 \rrbracket &= \llbracket e_1 \rrbracket \times_{\mathbb{N}} \llbracket e_2 \rrbracket \\
\llbracket e_1 < e_2 \rrbracket &= \llbracket e_1 \rrbracket <_{\mathbb{N}} \llbracket e_2 \rrbracket \\
\llbracket \text{iszero } e \rrbracket &= \text{zero}_{\mathbb{N}} \llbracket e \rrbracket \\
\llbracket \text{not } e \rrbracket &= \neg \llbracket e \rrbracket
\end{aligned}$$

En donde  $n \in \mathbb{N}$  es un natural y las constantes **False**, **True**  $\in \text{Bool}$ .

Los operadores  $+_{\mathbb{N}}$ ,  $\times_{\mathbb{N}}$ ,  $<_{\mathbb{N}}$  y  $\text{zero}_{\mathbb{N}}$  representan las funciones de suma, producto, menor y el *test* de cero definidas para los naturales respectivamente, mientras que  $\neg$  es la negación lógica para  $\text{Bool}$ <sup>a</sup>.

---

<sup>a</sup>Definición extraída de [1], [5]. [12], [38] y [39].

A partir de la definición anterior se puede implementar un intérprete para expresiones del lenguaje EAL mediante una función recursiva **eval** como sigue:

$$\begin{aligned}
\text{eval } n &= n \\
\text{eval true} &= \text{True} \\
\text{eval false} &= \text{False} \\
\text{eval } (e_1 + e_2) &= \text{eval } e_1 + \text{eval } e_2 \\
\text{eval } (e_1 * e_2) &= \text{eval } e_1 * \text{eval } e_2 \\
\text{eval } (e_1 < e_2) &= (\text{eval } e_1) < (\text{eval } e_2) \\
\text{eval } (\text{not } e) &= \text{not } (\text{eval } e) \\
\text{eval } (\text{iszero } e) &= (\text{eval } e) == 0
\end{aligned}$$

Con esta definición ahora podemos evaluar algunas expresiones pertenecientes a nuestro lenguaje EAL.

**Ejercicio 2.4.** Evalúa la expresión:  $\text{not } 3 < 5 + 7$

$$\begin{aligned}
\text{eval}(\text{not } 3 < 5 + 7) &= \\
\neg (\text{eval}(3 < 5 + 7)) &= \\
\neg (\text{eval}(3) < \text{eval}(5 + 7)) &= \\
\neg (\text{eval}(3) < \text{eval}(5) + \text{eval}(7)) &= \\
\neg (3 < 5 + 7) &= \\
\neg (3 < 12) &= \\
\neg (\text{True}) &= \\
\text{False}
\end{aligned}$$

**Ejercicio 2.5.** Evalúa la expresión:  $3 \times \text{True} > 8$

```
eval( 3 x True > 8 ) =  
eval( 3 x True ) > eval(8) =  
eval(3) x eval(True) > 8 =  
3 x true > 8 = error  
Expresión del lenguaje EAB mal formada.
```

**Definición 2.3.** Sea  $G$  una gramática libre de contexto, decimos que  $G$  es una gramática ambigua si existe una cadena para la cual se pueda tener más de una derivación a la izquierda<sup>a</sup>.

<sup>a</sup>Definición extraída de [5], [12], [40] y [41].

**Ejercicio 2.6.** ¿La gramática definida para EAL es ambigua? Argumenta tu respuesta. Sí porque podemos derivar de dos formas distintas la cadena:  $3 + 4 * 5$

```
e := e+e →  
      3 + e →  
      3 + e * e →  
      3 + 4 * e →  
      3 + 4 * 5.
```

```
e := e*e →  
      e * 5 →  
      e + e * 5 →  
      e + 4 * 5 →  
      3 + 4 * 5.
```

**Ejercicio 2.7.** ¿Cómo eliminarías la ambigüedad de esta gramática?

Con la introducción de paréntesis para marcar la precedencia de las operaciones. En el ejemplo anterior la ambigüedad queda eliminada en su totalidad:

$(3 + (4 * 5))$  ó  $((3 + 4) * 5)$

### 3 Clasificación de los lenguajes de programación

En computación existen diferentes clasificaciones que nos permiten agrupar lenguajes según sus características principales, como los enlistados a continuación.

Lenguajes compilados que precisan de un compilador para traducir el código de alto nivel a instrucciones de máquina que el procesador pueda ejecutar. Generalmente suele ser mas lento ya que requieren de un sistema sofisticado de inferencia de tipos, optimizaciones en la traducción de las instrucciones y en ocasiones entornos especiales para la ejecución de

los programas como la máquina virtual de Java (*JVM* por sus siglas en inglés)<sup>4</sup>.

Lenguajes interpretados, en donde las instrucciones son mapeadas uno a uno con la traducción a lenguaje de máquina y suelen ser menos robustos a la hora de encontrar errores de tipo, errores en la tipografía, caracteres faltantes, etc. Sin embargo permiten el modelado y la realización de programas mucho más rápido<sup>5</sup>.

También se pueden clasificar por paradigma como en el caso de los lenguajes funcionales que se enfocan en la especificación de un problema mediante condiciones en lugar de la solución teniendo la siguiente definición: "La programación funcional es un paradigma de programación en el que los programas se construyen aplicando y componiendo funciones. Es un paradigma de programación declarativa en el que las definiciones de funciones son árboles de expresiones que asignan valores a otros valores, en lugar de una secuencia de declaraciones imperativas que actualizan el estado de ejecución del programa"<sup>6</sup>.

En contraste a este paradigma existe el conocido como programación imperativa que se define como: "un paradigma de programación de software que utiliza declaraciones que cambian el estado de un programa. De la misma manera que el modo imperativo en los lenguajes naturales expresa órdenes, un programa imperativo consta de órdenes que debe ejecutar la computadora. La programación imperativa se centra en describir cómo funciona un programa paso a paso<sup>7</sup> en lugar de descripciones de alto nivel de sus resultados esperados"<sup>8</sup>.

El término se utiliza con frecuencia como un contrapunto de la programación declarativa, siendo el primero un conjunto de instrucciones para solucionar un determinado problema mientras que el segundo se centra en la especificación del problema mismo.

Otra clasificación que es muy socorrida en la computación es la llamada orientada a objetos que modela las entidades de un programa con su identidad, métodos, variables y que brindan características deseables como la herencia, el polimorfismo y el encapsulamiento de datos.

Esta clasificación pertenece al paradigma imperativo dado que se trabaja estrechamente con el estado del programa en cada paso de la ejecución, sus direcciones de memoria y sus sentencias de control<sup>9</sup>.

Por último mencionaremos brevemente la clasificación de los lenguajes lógicos que se puede definir como: "Un programa, base de datos o base de conocimientos en un lenguaje de programación lógica es un conjunto de oraciones en forma lógica que expresan hechos y

---

<sup>4</sup>Definición extraída de [43], [44], [45], [46]

<sup>5</sup>Definición extraída de [43], [44], [45], [46]

<sup>6</sup>Definición extraída de [51] y [52]

<sup>7</sup>[53]

<sup>8</sup>Extraído de [53], [54], [55] y [56]

<sup>9</sup>Definición extraída de [47], [48], [49]

reglas sobre un dominio específico”<sup>10</sup>.

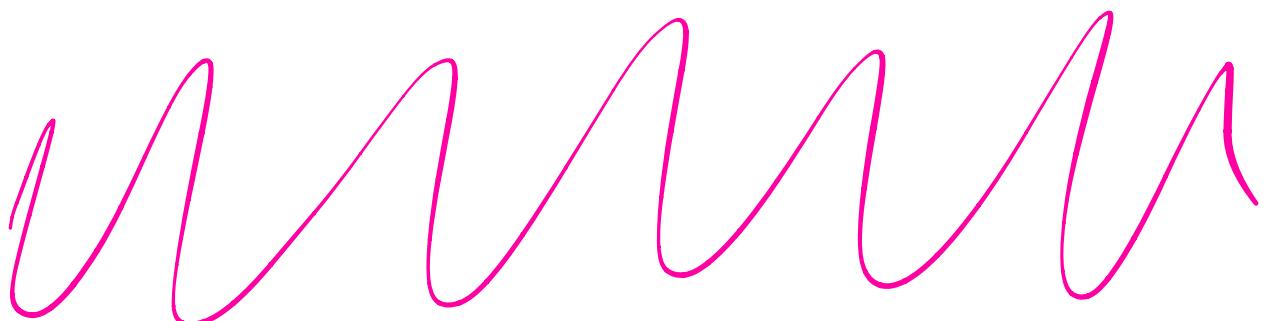
Generalmente este tipo de programas se apoyan definiendo predicados y objetos a los cuales estos predicados pueden ser aplicados así como instrucciones y operadores lógicos para dar solución a los problemas planteados en este modelo.

**Ejercicio 3.1.** Da una implementación del algoritmo *selection sort* en programación imperativa.

```
void swap(int *xp, int *yp){  
    int temp = *xp;  
    *xp = *yp;  
    *yp = temp;  
}  
  
void selectionSort(int arr[], int n){  
    int i, j, min_idx;  
    for (i = 0; i < n-1; i++){  
        min_idx = i;  
        for (j = i+1; j < n; j++)  
            if (arr[j] < arr[min_idx])  
                min_idx = j;  
        swap(&arr[min_idx], &arr[i]);  
    }  
}
```

**Ejercicio 3.2.** Da una implementación del algoritmo *selection sort* en programación declarativa.

```
ssort :: Ord t => [t] -> [t]  
ssort [] = []  
ssort xs = let { x = minimum xs }  
           in x : ssort (delete x xs)
```



---

<sup>10</sup>Definición extraída de [57]

## 4 Ejercicios para el lector

**Ejercicio 4.1.** El uso del carácter especial ';' al terminar una instrucción de programa en lenguajes de programación como C y Java ¿a qué tipo de regla corresponde (sintáctica, semántica o pragmática)?

**Ejercicio 4.2.** Ejemplifica la aplicación de la pragmática aplicada en el contexto de los lenguajes de programación.

**Ejercicio 4.3.** En el contexto de los lenguajes de programación muchas veces la semántica de una expresión es explicada en lenguaje natural en la documentación del lenguaje.

Explica por qué esto no es una semántica formal.

**Ejercicio 4.4.** Escribe una grámática para el alfabeto  $\Sigma = \{a, b\}$  y el lenguaje

$$L = \{a^n b^m \mid n > 0, m \geq n\}$$

**Ejercicio 4.5.** Escribe una gramática para el alfabeto  $\Sigma = \{a, b\}$  y el lenguaje

$$L = \text{cadenas palíndromas}$$

**Ejercicio 4.6.** Escribe una gramática para el alfabeto  $\Sigma = \{(, )\}$  y el lenguaje

$$L = \text{cadenas de paréntesis balanceados}$$

**Ejercicio 4.7.** Evalúa la expresión del lenguaje EAL.

$$7 * 8 < 3 * 9$$

**Ejercicio 4.8.** Evalúa la expresión del lenguaje EAL.

$$\text{not(iszero}((7 * 4) + (1 * 0)))$$

**Ejercicio 4.9.** Escribe el algoritmo de la búsqueda binaria en un arreglo ordenado en el paradigma de programación imperativo.

**Ejercicio 4.10.** Escribe el algoritmo de la búsqueda binaria en un arreglo ordenado en el paradigma de programación declarativo.

## Capítulo 2

# Herramientas matemáticas



Para estudiar a los lenguajes de programación es necesario definir una estructura que nos permita capturar su esencia matemática y nos proporcione un mecanismo para demostrar características y propiedades asociadas a éstos.

Una forma de definir formalmente un lenguaje de programación es mediante el uso de juicios lógicos<sup>1</sup>, para definir propiedades, pertenencia de los elementos de nuestra estructura a un conjunto, relación entre diferentes elementos de una misma estructura, etc. Que

---

<sup>1</sup>En lógica el equivalente a un juicio lógico son los predicados que denominan una característica que se cumple en una proposición

junto con la aplicación de reglas de inferencia nos permitirán dar una definición inductiva de las estructuras que pretendemos estudiar<sup>2</sup>.

## Objetivo

Visitar las estructuras matemáticas que hemos estudiado con anterioridad, en particular listas, árboles, reglas de inferencia y estructuras inductivas para familiarizarnos con la naturaleza recursiva que éstas poseen.

Este tipo de mecanismos es particularmente útil para ilustrar las propiedades de los lenguajes que definiremos a lo largo de los capítulos de este manual.

## Planteamiento

En este capítulo exploraremos diferentes estructuras empezando por los juicios lógicos que constituyen las entidades fundamentales sobre las cuales iniciaremos el estudio de los lenguajes. También estudiaremos brevemente estructuras inductivas como cadenas y listas para enunciar el principio de inducción que nos permite demostrar propiedades sobre dichas estructuras.

Por último estudiaremos un sistema de transición para modelar los estados y reglas de evaluación de la instancia de un juego<sup>3</sup>.

# 1 Juicios lógicos

### Ejercicio 1.1. ¿Qué es un objeto?

"En el lenguaje habitual de las matemáticas, un objeto es cualquier cosa que ha sido (o podría ser) definida formalmente y con la que se pueden realizar razonamientos deductivos y pruebas matemáticas"<sup>a</sup>.

---

<sup>a</sup>Definición extraída de [58], [59] y [60].

### Ejercicio 1.2. ¿Qué es un juicio?

"En lógica matemática, un juicio es una afirmación o enunciación en un metalinguaje de una característica o propiedad sobre un objeto o elemento de un dominio"<sup>a</sup>.

---

<sup>2</sup>Para las personas que poseen una madurez matemática superior a aquella que la que un estudiante de cuarto semestre de la Facultad de Ciencias pudiera tener se recomienda encarecidamente iniciar el estudio de este manual en el capítulo 3: Sintáxis. Los temas recomendados para omitir los primeros dos capítulos del manual son: estructuras discretas, juicios lógicos, reglas inductivas, reglas de inferencia e inducción estructural.

<sup>3</sup>Estos sistemas de transición se estudiaran a detalle mas adelante en este manual.

<sup>a</sup>Definición extraída de [61] y [62].

**Ejercicio 1.3.** Proporciona un listado de juicios sobre objetos matemáticos (Puedes suponer definidos con anterioridad elementos como  $\mathbb{N}$ ,  $\mathbb{Q}$ ,  $\text{Bool}$ ,  $\text{String}$ , funciones, relaciones de orden, operadores aritméticos, etc).

”Hola Mundo” $\text{String}$	La cadena ”Hola Mundo” es de tipo $\text{String}$ .
$L \ NP$	El lenguaje $L$ es $NP$ .
$a > b$	el número $a$ es más grande que el número $b$ .
$A \mid B$	El evento $A$ es independiente al evento $B$ .
$\text{True} \ \text{Bool}$	La constante $\text{True}$ es de tipo $\text{Bool}$ .
$h = g \circ f$	la función $h$ es la composición de la función $g$ con la función $f$ .
$3.1416\dots \ \mathbb{I}$	El número $3.1416\dots$ es irracional
$0 \ \mathbb{N}$	$0$ es un Natural.
$S(0) \ \text{impar}$	El sucesor de $0$ es impar.
$\frac{p}{q} \ \mathbb{Q}$	La fracción $\frac{p}{q}$ es un racional

## 2 Definiciones inductivas, reglas de inferencia e inducción estructural

Las reglas de inferencia<sup>4</sup> están compuestas por dos secciones: una sección superior y una inferior. La sección superior es en donde se agrupan las condiciones necesarias para que la regla pueda ser aplicada (esta sección es la que contiene las premisas o hipótesis), cada una de éstas están separadas por una coma (’,’) y todas deben de cumplirse<sup>5</sup>. En ocasiones un conjunto de reglas de inferencia pueden ser también reglas inductivas si la conclusión proporciona un nuevo elemento que pertenece a la misma categoría o posee la misma propiedad que las premisas, a dicha definición se lo conoce como una definición inductiva.

La sección inferior de una regla de inferencia enlista el resultado de la aplicación de la regla a las premisas. Ocasionalmente la regla tendrá un nombre que ayuda a identificar cuál de ellas es aplicada cuando se trabaja con derivaciones y razonamientos. En esta sección de la regla puede haber uno o más elementos resultados de la aplicación pero no puede ser vacía.

Las reglas inductivas reciben su nombre dado que la aplicación de dicha regla proporciona

<sup>4</sup>En esta manual utilizaremos el término ”regla de inferencia” o ”juicio lógico” de forma indistinta, aunque es importante notar que un conjunto de reglas de inferencia no necesariamente define una definición inductiva.

<sup>5</sup>El signo de puntuación (’,’) se puede interpretar como el operador  $\wedge$  donde cada hipótesis es un argumento de dicho operador.

un elemento de las mismas características que las premisas. Este tipo de regla permite realizar razonamientos inductivos partiendo de un elemento hasta llegar a un axioma.

Los axiomas son las reglas de inferencia que carecen de hipótesis dado que siempre serán verdad y no es necesario suponer nada para concluir las<sup>6</sup>.

Las estructuras inductivas son entonces aquellas estructuras que podemos construir a partir de la aplicación de las reglas inductivas que a su vez son reglas de inferencia. Las estructuras recursivas por el contrario, son estructuras que se generan utilizando una llamada al elemento más primitivo que puede formar parte de dicha estructura (muchas veces este elemento constituye la cláusula de escape para la recursión inducida por la definición y constituiría un axioma en su representación como regla inductiva) o una llamada a un constructor con un elemento nuevo y la estructura misma que se desea definir como argumentos<sup>7</sup>.

Por último mencionaremos brevemente el principio de inducción para estructuras recursivas. Este principio consiste en la aplicación de la propiedad de inducción asociada a las estructuras que estudiaremos en los ejercicios para demostrar cualidades que se cumplen para cualquier instancia.

El principio de inducción funciona de la siguiente manera:

- A) Se demuestra la validez de la proposición para los elementos más básicos de la estructura (que serán aquellos generados a partir de axiomas).
- B) Posteriormente se asume que la propiedad es válida para cualquier instancia.
- C) Por último se demuestra la propiedad aplicada para los constructores que generan instancias más grandes combinando elementos que ya pertenecían a la estructura aplicando la hipótesis del inciso anterior.

**Definición 2.1.** Sea  $A = \{0, 1\}$  un conjunto. Definimos el tipo de dato recursivo  $A^*$  (cadenas binarias de tipo A) como sigue:

$$\frac{}{\varepsilon \in A^*} \text{ (eps)} \quad \frac{a \in A, s \in A^*}{\langle a, s \rangle \in A^*} \text{ (tup)}$$

la regla *eps* se lee como: "la cadena vacía es una cadena binaria de tipo A" y la regla *tup* se lee como: "si a es de tipo A y s es una cadena binaria de tipo A, entonces la construcción de una lista con cabeza a y cola s es una cadena binaria de tipo A"<sup>a</sup>.

Por ejemplo, la cadena 10101 se representa por la 5-tupla  $\langle 1, \langle 0, \langle 1, \langle 0, \langle 1, \epsilon \rangle \rangle \rangle \rangle \rangle$

Definimos la concatenación de cadenas binarias (denotado por el operador " $\bullet$ ")

<sup>6</sup>Definición formulada de [1], [5], [9], [63], [64] y [65].

<sup>7</sup>Definición formulada a partir de [63] y [64]

mediante la siguiente función:

Caso base: ( $\text{eps}$ )       $\varepsilon \bullet t = t$       Constructor: ( $\text{tup}$ )       $\langle a, s \rangle \bullet t = \langle a, \langle s \bullet t \rangle \rangle$

<sup>a</sup>Definición formulada a partir de [67], [68] y [69]

**Ejercicio 2.1.** Utiliza la inducción estructural para demostrar:

$$t \bullet \varepsilon = t$$

### Caso Base $t = \varepsilon$

$$\varepsilon \bullet \varepsilon = \varepsilon$$

(Por la definición de  $\bullet$ )

$$\varepsilon = t.$$

**Hipótesis Inductiva**  $t \bullet \varepsilon = t$  para toda  $t \in A^*$

**Paso Inductivo** por demostrar  $\langle a, t \rangle \bullet \varepsilon = \langle a, t \rangle$  para toda  $a \in A$  y  $t \in A^*$

$$\langle a, t \rangle \bullet \varepsilon = \langle a, \langle t \bullet \varepsilon \rangle \rangle \quad (\text{Por la definición de } \bullet)$$

$$\langle t \bullet \varepsilon \rangle = t$$

(Por la hipótesis inductiva)

$$\langle a, t \rangle \bullet \varepsilon = \langle a, \langle t \bullet \varepsilon \rangle \rangle = \langle a, t \rangle$$

**Ejercicio 2.2.** B) utiliza las reglas (*eps*) y (*tup*) para derivar las cadenas:

$$\langle 1, \langle 0, \langle 0, \varepsilon \rangle \rangle \rangle, \quad \langle 1, \langle 0, \langle 0 \langle 1, \varepsilon \rangle \rangle \rangle \rangle \quad y \quad \langle 1, \langle 1, \langle 1, \langle 0, \langle 1, \langle 0, \varepsilon \rangle \rangle \rangle \rangle \rangle \rangle$$

$$\frac{\frac{\frac{0 \in A}{(ax)} \frac{(ax) \quad \varepsilon \in A^*}{\langle 0, \rangle \in A^*} (eps)}{\langle 0, \rangle \in A^*} (tup)}{\frac{0 \in A}{(ax)} \frac{\langle 0, \langle 0, \varepsilon \rangle \rangle \in A^*}{\langle 1, \langle 0, \langle 0, \varepsilon \rangle \rangle \rangle \in A^*} (tup)}$$

$$\frac{\frac{\frac{\frac{1 \in A}{\overline{1 \in A}} (ax) \frac{\varepsilon \in A}{\overline{\varepsilon \in A}} (eps)}{\langle 1, \varepsilon \rangle \in A^*} (tup)}{\frac{0 \in A}{\overline{0 \in A}} (ax) \langle 0 \langle 1, \varepsilon \rangle \rangle \in A^*} (tup)}{\frac{0 \in A}{\overline{1 \in A}} (ax) \langle 0, \langle 0 \langle 1, \varepsilon \rangle \rangle \rangle \in A^*} (tup)$$

$$\begin{array}{c}
\frac{}{\overline{0 \in A} \quad (ax)} \frac{}{\overline{1 \in A} \quad (ax)} \frac{\overline{0 \in A} \quad (ax) \quad \overline{\varepsilon \in A^*} \quad (eps)}{\langle 0, \varepsilon \rangle \in A^* \quad (tup)} \\
\frac{}{\overline{0 \in A} \quad (ax)} \frac{}{\overline{1 \in A} \quad (ax)} \frac{\overline{0 \in A} \quad (ax) \quad \overline{\langle 1, \langle 0, \varepsilon \rangle \rangle \in A^*} \quad (tup)}{\langle 1, \langle 0, \varepsilon \rangle \rangle \in A^* \quad (tup)} \\
\frac{}{\overline{1 \in A} \quad (ax)} \frac{}{\overline{1 \in A} \quad (ax)} \frac{\overline{0 \in A} \quad (ax) \quad \overline{\langle 1, \langle 1, \langle 0, \varepsilon \rangle \rangle \rangle \in A^*} \quad (tup)}{\langle 1, \langle 1, \langle 0, \varepsilon \rangle \rangle \rangle \in A^* \quad (tup)} \\
\frac{}{\overline{1 \in A} \quad (ax)} \frac{}{\overline{1 \in A} \quad (ax)} \frac{\overline{0 \in A} \quad (ax) \quad \overline{\langle 1, \langle 1, \langle 1, \langle 0, \varepsilon \rangle \rangle \rangle \rangle \in A^*} \quad (tup)}{\langle 1, \langle 1, \langle 1, \langle 0, \varepsilon \rangle \rangle \rangle \rangle \in A^* \quad (tup)} \\
\frac{}{\overline{1 \in A} \quad (ax)} \frac{}{\overline{1 \in A} \quad (ax)} \frac{\overline{0 \in A} \quad (ax) \quad \overline{\langle 1, \langle 1, \langle 1, \langle 1, \langle 0, \varepsilon \rangle \rangle \rangle \rangle \rangle \in A^*} \quad (tup)}{\langle 1, \langle 1, \langle 1, \langle 1, \langle 0, \varepsilon \rangle \rangle \rangle \rangle \rangle \in A^* \quad (tup)}
\end{array}$$

**Definición 2.2.** Definimos la función para contar caracteres en una cadena denotada por  $n_c$ <sup>a</sup> como sigue:

$$n_c(\langle a, t \rangle) = \begin{cases} n_c(\varepsilon) = 0 & \text{Sí } a = c \\ 1 + n_c(t) & \text{Sí } a \neq c \\ n_c(t) & \end{cases}$$

<sup>a</sup>Definición formulada a partir de [67], [68] y [69]

**Ejercicio 2.3.** Utiliza la inducción estructural para demostrar

$$n_c(s \bullet t) = n_c(s) + n_c(t)$$

**Caso Base**  $s = \varepsilon$

$$\begin{aligned} n_c(\varepsilon \bullet t) &= n_c(t) && \text{(Por la definición de } \bullet\text{)} \\ &= 0 + n_c(t) = n_c(\varepsilon) + n_c(t) && \text{(Por la definición de } n_c\text{)} \\ &= n_c(s) + n_c(t) \end{aligned}$$

**Hipótesis Inductiva**  $n_c(s \bullet t) = n_c(s) + n_c(t)$

**Paso Inductivo** Por demostrar para  $s = \langle a, s' \rangle$

$$n_c(\langle a, s' \rangle \bullet t) = n_c(\langle a, s' \bullet t \rangle) \quad \text{(Por la definición de } \bullet\text{)}$$

**Caso 1**  $a = c$  entonces tenemos

$$\begin{aligned} n_c(\langle a, s' \bullet t \rangle) &= 1 + n_c(s' \bullet t) && \text{(Por la definición de } n_c\text{)} \\ &= 1 + n_c(s') + n_c(t) && \text{(Por la hipótesis inductiva)} \\ &= (1 + n_c(s')) + n_c(t) = n_c(\langle a, s' \rangle) + n_c(t) && \text{(Por la definición de } n_c\text{)} \end{aligned}$$

**Caso 2**  $a \neq c$  entonces tenemos

$$\begin{aligned} n_c(\langle a, s' \bullet t \rangle) &= n_c(s' \bullet t) && \text{(Por la definición de } n_c\text{)} \\ &= n_c(s') + n_c(t) && \text{(Por la hipótesis inductiva)} \\ &= n_c(\langle a, s' \rangle) + n_c(t) && \text{(Por la definición de } n_c\text{)} \end{aligned}$$

Formato

**Definición 2.3.** Definimos las funciones  $rev$  y  $len$  para obtener la reversa de una cadena binaria y la longitud respectivamente como<sup>a</sup>:

$$rev(\varepsilon) = \epsilon \quad rev(\langle a, s \rangle) = rev(s) \bullet \langle a, \varepsilon \rangle$$

$$len(\varepsilon) = 0 \quad len(\langle a, s \rangle) = 1 + len(s)$$

<sup>a</sup>Definición formulada a partir de [67] y [70]

**Ejercicio 2.4.** Utiliza inducción estructural para demostrar

$$len(s \bullet t) = len(s) + len(t)$$

**Caso Base**  $s = \varepsilon$

$$\begin{aligned}
 \text{len}(\varepsilon \bullet t) &= \text{len}(t) && \text{(Por la definición de } \bullet\text{)} \\
 &= 0 + \text{len}(t) = \text{len}(\varepsilon) + \text{len}(t) && \text{(Por la definición de } \text{len}\text{)} \\
 &= \text{len}(s) + \text{len}(t)
 \end{aligned}$$

**Hipótesis inductiva**  $\text{len}(s \bullet t) = \text{len}(s) + \text{len}(t)$

**Paso Inductivo** Por demostrar para  $s = \langle a, s' \rangle$

$$\begin{aligned}
 \text{len}(\langle a, s' \rangle \bullet t) &= \text{len}(\langle a, \langle s' \bullet t \rangle \rangle) && \text{(Por la definición } \bullet\text{)} \\
 &= 1 + \text{len}(s' \bullet t) = 1 + \text{len}(s') + \text{len}(t) && \text{(Por la hipótesis inductiva)} \\
 &= (1 + \text{len}(s')) + \text{len}(t) = \text{len}(\langle a, s' \rangle) + \text{len}(t)
 \end{aligned}$$

**Ejercicio 2.5.** Utiliza inducción estructural para demostrar

$$\text{len}(\text{rev}(s)) = \text{len}(s)$$

**Caso Base**  $s = \varepsilon$

$$\begin{aligned}
 \text{len}(\text{rev}(\varepsilon)) &= \text{len}(\varepsilon) && \text{(Por la definición de } \text{rev}\text{)} \\
 \text{len}(\text{rev}(s)) &= \text{len}(\text{rev}(\varepsilon)) = \text{len}(\varepsilon) = \text{len}(s).
 \end{aligned}$$

**Hipótesis Inductiva**  $\text{len}(\text{rev}(s)) = \text{len}(s)$

**Paso Inductivo**  $s = \langle a, s \rangle$

$$\begin{aligned}
 \text{len}(\text{rev}(\langle a, s \rangle)) &= \text{len}(\text{rev}(s) \bullet \langle a, \varepsilon \rangle) && \text{(Por la definición de } \text{rev}\text{)} \\
 \text{len}(\text{rev}(s) \bullet \langle a, \varepsilon \rangle) &= \text{len}(\text{rev}(s)) + \text{len}(\langle a, \varepsilon \rangle) && \text{(Por la propiedad anterior)} \\
 \text{len}(\text{rev}(s)) + 1 + \text{len}(\varepsilon) &= \text{len}(\text{rev}(s)) + 1 + 0 \\
 1 + \text{len}(\text{rev}(s)) &= 1 + \text{len}(s) && \text{(Por la hipótesis inductiva)} \\
 \text{len}(\langle a, s \rangle) &\text{ por la definición de } \text{len} && \text{(Por definición de } \text{len}\text{)}
 \end{aligned}$$

**Definición 2.4.** Definimos los árboles binarios de tipo A con las siguientes reglas<sup>a</sup>:

1.  $\emptyset$  es un árbol de tipo A.
2. Si  $t_1$  y  $t_2$  son árboles de tipo A y  $a \in A$  entonces  $\text{node}(a, t_1, t_2)$  es un árbol binario de tipo A.

Decimos que un árbol de altura  $k$  es balanceado si:

- 1)  $t = \emptyset$  y  $k = 0$  ó
- 2)  $t = \text{node}(a, t_1, t_2)$  de altura  $k$  donde  $t_1$  y  $t_2$  son árboles balanceados de altura  $k-1$ .

Definimos la función que cuenta el número de nodos denotada  $n(t)$  como:

$$\begin{aligned}
 n(\emptyset) &= 0 \\
 n(\text{node}(a, t_1, t_2)) &= 1 + n(t_1) + n(t_2)
 \end{aligned}$$

<sup>a</sup>Definición formulada a partir de [67], [68] y [69]

**Ejercicio 2.6.** Demuestra utilizando inducción estructural que si  $t$  es un árbol balanceado de tipo A con altura  $k$  entonces  $n(t) = 2^k - 1$

**Caso Base**  $t = \emptyset, k = 0$

$$n(\emptyset) = 2^0 - 1 = 1 - 1 = 0.$$

**Hipótesis Inductiva**  $n(t) = 2^k - 1$  para todo árbol balanceado  $t$  donde  $k$  es la altura del mismo.

**Paso Inductivo** Por demostrar para  $t = \text{node}(a, t_1, t_2)$  donde  $t_1, t_2$  son árboles balanceados de altura  $k - 1$  se cumple que  $n(a, t_1, t_2) = 2^k - 1$  donde  $k$  es la altura asociada al árbol  $t$ .

$$\begin{aligned} n(\text{node}(a, t_1, t_2)) &= 1 + n(t_1) + n(t_2) && (\text{Por definición de } n(t)) \\ &= 1 + (2^{k-1} - 1) + (2^{k-1} - 1) && (\text{Por la hipótesis inductiva}) \\ &= 1 + 2(2^{k-1} - 1) = 2^k - 2 + 1 = 2^k - 1 \end{aligned}$$

### 3 Sistemas de transición

En el contexto de las ciencias de la computación los sistemas de transición resultan particularmente útiles para obtener cualquier configuración de un programa en cada paso de su ejecución en forma de estados. En particular tendremos dos estados especiales para denotar la configuración inicial y la configuración final y nos apoyaremos de un conjunto de reglas que nos permiten aplicar instrucciones y operadores a los estados.

Estos sistemas resultan particularmente útiles ya que encapsulan la definición de un problema a partir del cual podemos obtener todas las configuraciones posibles partiendo del estado inicial al estado final facilitando el análisis de la complejidad en tiempo y espacio del mismo.

Para el enfoque que seguimos en este manual definiremos las reglas de nuestros sistemas de transición como juicios lógicos.

Estructuras similares se han estudiado en cursos como autómatas y lenguajes formales<sup>8</sup> donde adicionalmente se definía el alfabeto y se consideraba la lectura de un carácter en cada transición. En este caso omitiremos estos elementos y nos concentraremos únicamente en los estados y las reglas de transición.

<sup>8</sup>Conforme al plan de estudios que se imparte desde el 2013 en la Facultad de Ciencias de la Universidad Nacional Autónoma de México con clave de asignatura 1425.

**Ejercicio 3.1.** Supón que se quiere modelar una partida de pinpón donde hay 2 jugadores A y B (donde el jugador A siempre empieza primero), y un marcador que cuenta el número de puntos de A y B representado como: jugador en turno, Puntuación de A, Puntuación de B.

De tal forma que cada estado indica la información del juego hasta ese punto. Por ejemplo si el jugador B anota, el estado se modelaría como  $(B, X, Y) \rightarrow (B, X, Y+1)$  ya que el jugador que anota un punto vuelve a tirar la pelota en el siguiente turno.

La partida acaba cuando cualquiera de los dos anota 10 puntos.

A) Define formalmente el conjunto de estados (índica estados iniciales y finales).

$$I = \{(A, 0, 0)\}$$

$$\Gamma = \{C, X, Y\} \text{ donde } C \in \{A, B\} \text{ y } 0 \leq X, Y \leq 10$$

$$F = \{C, X, Y\} \text{ donde } C \in \{A, B\} \text{ y } X \text{ ó } Y = 10$$

} equation

B) Define la función de transición

$$\text{El jugador A puntuá: } f((A, X, Y)) = (A, X+1, Y)$$

$$\text{El jugador B puntuá: } f((B, X, Y)) = (B, X, Y+1)$$

$$\text{A no puntuá: } f((A, X, Y)) = (B, X, Y)$$

$$\text{B no puntuá: } f((B, X, Y)) = (A, X, Y)$$

C) Define con reglas de inferencia la función de transición escrita en el inciso anterior.

$$\begin{array}{c} \frac{(A, X, Y) \text{ Estado}, (A, X + 1, Y) \text{ Estado}}{(A, X, Y) \rightarrow (A, X + 1, Y) \text{ Estado}} \quad (A+) \quad \frac{(B, X, Y) \text{ Estado}, (B, X, Y + 1) \text{ Estado}}{(B, X, Y) \rightarrow (B, X, Y + 1) \text{ Estado}} \quad (B+) \\ \frac{(A, X, Y) \text{ Estado}, (B, X, Y) \text{ Estado}}{(A, X, Y) \rightarrow (B, X, Y) \text{ Estado}} \quad (A-) \quad \frac{(B, X, Y) \text{ Estado}, (B, X, Y + 1) \text{ Estado}}{(B, X, Y) \rightarrow (B, X, Y) \text{ Estado}} \quad (B-) \end{array}$$

} inferencia

D) Muestra una ejecución donde gane el jugador B especificando cada estado desde el inicial hasta el final.

$$(A, 0, 0) \rightarrow (A, 1, 0) \rightarrow (A, 1, 0) \rightarrow (B, 1, 1) \rightarrow (B, 1, 2) \rightarrow (B, 1, 3) \rightarrow (B, 1, 4) \rightarrow (B, 1, 4) \rightarrow (A, 1, 4) \rightarrow (B, 1, 5) \rightarrow (B, 1, 5) \rightarrow (B, 1, 6) \rightarrow (B, 1, 6) \rightarrow (A, 2, 6) \rightarrow (A, 2, 6) \rightarrow (B, 2, 7) \rightarrow (B, 2, 8) \rightarrow (B, 2, 8) \rightarrow (A, 3, 8) \rightarrow (A, 4, 8) \rightarrow (A, 5, 8) \rightarrow (A, 5, 8) \rightarrow (B, 5, 9) \rightarrow (B, 5, 10)$$

} accion de B

En donde la secuencia se puede interpretar como: "El jugador A hizo el saque inicial" después "El jugador A anotó" después "El jugador A hizo el saque inicial pues anotó en el turno anterior" desoués "El jugador B anotó" después "El jugador B hizo el saque inicial pues anotó en el turno anterior" después "El jugador B anotó" ...

## 4 Ejercicios para el lector

**Definición 4.1.** Definimos la función  $\text{map } f$  que recibe una función  $f$  y una cadena  $s$  de la siguiente forma<sup>a</sup>:

$$\begin{aligned}\text{map } f(\varepsilon) &= \varepsilon \\ \text{map } f(\langle a, s \rangle) &= \langle f a, \text{map } f s \rangle\end{aligned}$$

<sup>a</sup>Definición formulada a partir de [67], [68] y [69]

**Ejercicio 4.1.** Demuestra utilizando la inducción estructural sobre nuestras cadenas lo siguiente:

$$\text{len}(\text{map } f \langle a, s \rangle) = \text{len} (\langle a, s \rangle)$$

**Ejercicio 4.2.** Demuestra utilizando la inducción estructural sobre nuestras cadenas para demostrar que

$$\text{map } f(\langle a, s \rangle \bullet \langle b, t \rangle) = \text{map } f \langle a, s \rangle \bullet \text{map } f \langle b, t \rangle$$

**Ejercicio 4.3.** Escribe una derivación para la cadena

$$\langle 1, \langle 0, \varepsilon \rangle \rangle$$

**Ejercicio 4.4.** Escribe una derivación para la cadena

$$\langle 1, \langle 1, \langle 1 \langle 0, \varepsilon \rangle \rangle \rangle \rangle$$

**Ejercicio 4.5.** Escribe una derivación para la cadena

$$\langle 1, \langle 0, \langle 0, \langle 0, \langle 0, \langle 1, \varepsilon \rangle \rangle \rangle \rangle \rangle \rangle$$

**Ejercicio 4.6.** Demuestra utilizando la inducción estructural sobre árboles binarios completos que se cumple lo siguiente

Sea  $T$  un árbol binario completo de altura  $h$  y número de nodos  $n$  se cumple que  $n \leq 2^{h+1} - 1$

**Ejercicio 4.7.** Demuestra utilizando inducción estructural sobre árboles binarios completos que lo siguiente se cumple

$$i(T) = \frac{n(T) - 1}{2}$$

donde  $i(T)$  es el número de nodos internos en un árbol y  $n(T)$  es el número de total de nodos.

**Ejercicio 4.8.** De acuerdo a la máquina de transición definida en el ejercicio 3.1 ¿los jugadores A y B pueden empatar?

Justifica tu respuesta

**Ejercicio 4.9.** De acuerdo a la máquina de transición definida en el ejercicio 3.1 muestra una ejecución en la que el jugador A gane.

**Ejercicio 4.10.** Suponga que se necesita definir un lenguaje que permita controlar un robot con movimientos y funcionalidades muy simples. El robot se mueve sobre una cuadricula siguiendo las instrucciones especificadas por el programa.

Al inicio el robot se encuentra en la coordenada  $(0, 0)$  y viendo hacia el norte. El programa consiste en una secuencia posiblemente vacía de los comandos **move** y **turn** separados por punto y coma, cada comando tiene el siguiente funcionamiento:

- **turn** hace que el robot dé un giro de 90 grados en el sentido de las manecillas del reloj.
- **move** provoca que el robot avance una casilla en la dirección hacia la que está viendo.

Donde un ejemplo válido puede ser el siguiente programa:

*move; turn; move; turn; turn; turn; move*

Al final del programa el robot termina en la casilla  $(2, 1)$ . La primera entrada de la coordenada indica la posición vertical mientras que la segunda es la posición horizontal.

Con la especificación discutida anteriormente contesta los siguientes puntos:

- Determina el conjunto de estados.
- Identifica los estados iniciales y finales del sistema de transición.
- Define la función de transición  $\rightarrow_R$  que indique como se debe transitar entre los estados del sistema. De tal forma que defina una semántica operacional de paso pequeño.
- Muestra paso a paso la ejecución del programa:

*move; turn; move; turn; turn; turn; move*

Utilizando la relación  $\rightarrow_R$  y partiendo del estado inicial.

Para finalizar esta sección vamos a revisar dos ejercicios prácticos que implementaremos en Haskell. Estos ejercicios combinan las definiciones de las funciones que hemos estudiado hasta este punto para listas y que exemplifican las propiedades recursivas de estas estructuras en un lenguaje de programación funcional<sup>9</sup>.

**Ejercicio 4.11.** La validación del número de una tarjeta de crédito se hace implementando un algoritmo **checkSum** obteniendo información de los dígitos que la componen.

---

<sup>9</sup>Los ejercicios 4.11 y 4.12 fueron extraídos de [75].

Para eso implementaremos nuestro propio validador de tarjetas de crédito como sigue:

Los dígitos que están una posición impar (empezando por el índice 0 de derecha a izquierda) se deben duplicar.

Posteriormente se suman todos los dígitos de los números que componen la tarjeta (aquellos números que fueron duplicados tendrán dos dígitos, los cuales deben de ser sumados junto con el resto que no se modificaron).

Por último se aplica el módulo 10 al resultado de la suma, si el resultado es diferente de 0 entonces la tarjeta es inválida, en caso contrario es válida.

1. Implementa la función `getDigits :: Int → [Int]`

Por ejemplo, la tarjeta 1348 1548 9998 6535 dará como resultado la lista: [1,3,4,8,1,5,4,8,9,9,9,8,6,3,5,3,5] (asumimos que la entrada son solo los dígitos de la tarjeta SIN separación).

2. Implementa la función `reverseDigits :: [Int] → [Int]`

Esta función obtiene la reversa de la lista de los dígitos obtenidos en el paso anterior (será usada como auxiliar).

3. Implementa la función `uplicateOddPositionDigits :: [Int] → [Int]`

Por ejemplo: la lista [1,3,4,8,1,5,4,8,9,9,9,8,6,3,5,3,5] solo duplicará los números que estén en una posición impar obteniendo como resultado: [1,6,4,16,1,10,4,16,9,18,9,16,6,6,5,6,5].

4. Implementa la función `toDigits :: Int → [Int]`

El objetivo de la implementación de esta función es el de separar los números duplicados mayores o iguales a 10 en sus dos dígitos componentes para poder sumarlos con el resto que se mantuvieron sin cambio, de tal forma que `toDigits([1,6,4,16,1,10,4,16,9,18,9,16,6,6,5,6,5]) = [1,6,4,1,6,1,10,4,1,6,9,1,8,9,1,6,6,6,5,6,5]`

5. Implementa la función `sumDigitsInList :: [Int] → Int`

Esta función simplemente suma los dígitos contenidos en la lista (puedes utilizar la función `sum` provista por GHCi).

7. Implementa la función `checkSumCreditCard :: Int → Bool`

Recuerda que el criterio de validéz es que la suma de los dígitos filtrados de la tarjeta módulo 10 sea igual a 0.

**Ejercicio 4.12.** Las torres de Hanoi son un rompecabezas clásico cuya solución puede ser escrita de forma recursiva. Los discos de diferentes tamaños se apilan del más grande al más pequeño y el objetivo es moverlos del pivote A al pivote C utilizando un pivote auxiliar B con  $n = 4$ .

Las reglas son las siguientes:

- Ningún disco puede estar encima de un disco más pequeño.
- Por movimiento solo es válido el desplazamiento de un disco hacia otro pivote.

Se definen los siguientes tipos para implementar la solución al puzzle de la siguiente forma:

Type Peg = String  
Type move = (Peg, Peg)

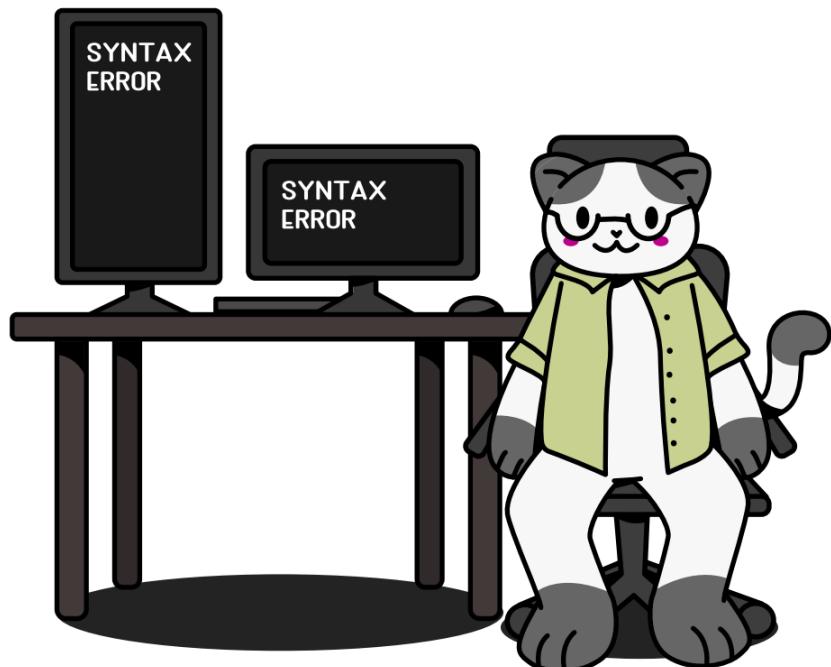
código

Implementa la función `Hanoi :: Int -> Peg -> Peg -> Peg -> [move]`

Que dada el número de discos y los pivotes regrese una lista con los movimientos para mover los discos del primer pivote al pivote objetivo, por ejemplo:  
`Hanoi 2 "a" "b" "c" == [("a", "c"), ("a", "b"), ("c", "b")]`

## Capítulo 3

# Sintáxis



La sintáxis de un lenguaje de programación permite delimitar el conjunto de las cadenas que pertenecen a él y al mismo tiempo constituye una herramienta de razonamiento para demostrar propiedades utilizando el principio de inducción aplicado a los juicios o reglas del lenguaje mismo. De la misma forma permite la definición de funciones (como la función de evaluación). Para esto se definen dos clasificaciones de la sintáxis de un lenguaje y se conocen como sintáxis concreta y sintáxis abstracta.

La sintáxis concreta que se relaciona con la representación de las cadenas del lenguaje y comúnmente es denotada por una gramática libre de contexto. Esta gramática está

pensada para el usuario del lenguaje, es decir una persona y se le denomina como un lenguaje de "alto nivel" para la humana comprensión del programa que se desea escribir o leer. Vamos a denotar las expresiones del lenguaje como " $E$ "<sup>1</sup>.

Otra clasificación es la sintaxis abstracta que se relaciona con la estructura del lenguaje y comúnmente es definida mediante reglas de inferencia para construir un árbol sintáctico donde cada nodo corresponde a una instrucción del lenguaje y los hijos a los parámetros que recibe dicha instrucción. De igual forma las hojas de los árboles sintácticos corresponden a los tipos primitivos del lenguaje (enteros, booleanos, caracteres).

En este capítulo vamos a denotar a un árbol de sintaxis abstracta como " $asa$ "<sup>2</sup>.

Los árboles sintácticos son útiles para definir la jerarquía de las operaciones a realizar durante la evaluación de una expresión bien formada de un lenguaje y eliminar la ambigüedad del mismo dado que un árbol de sintaxis abstracta es único sin importar la representación que una expresión tenga en sintaxis concreta.

Al proceso mediante el cual se obtiene el árbol de sintaxis abstracta de una expresión en sintaxis concreta se le conoce como análisis sintáctico, aquí se verifica que sea una expresión válida del lenguaje y mediante un conjunto de reglas o juicios se obtiene representación en sintaxis abstracta (es importante notar que no siempre será posible encontrarla) de tal forma que el siguiente juicio se cumple:  $e \ E \longleftrightarrow a \ asa$ <sup>3</sup>, donde esta relación se interpreta como " $a$  es el árbol de sintaxis abstracta asociado a la expresión  $e$ ". Por lo tanto, para definir esta relación para cualquier lenguaje de programación necesitamos construir los niveles sintácticos correspondientes para poder enunciar formalmente esta propiedad.

## Objetivo

Definir los diferentes niveles sintácticos para nuestro lenguaje **EAB**, mostrando las principales diferencias que existen entre la sintaxis concreta y abstracta así como estudiar cómo se relacionan ambos niveles definiendo entonces las reglas para implementar el analizador sintáctico de este lenguaje.

## Planteamiento

Iniciaremos el estudio de este capítulo brindando las definiciones correspondientes de sintaxis concreta y sintaxis abstracta.

Aprovecharemos también para definir el lenguaje con el que vamos a trabajar en los si-

<sup>1</sup>Definición formulada a partir de [1], [2], [5], [12] y [76].

<sup>2</sup>Definición formulada a partir de [5], [12], [77], [78] y [79].

<sup>3</sup>La definición de esta relación fue formulada partir de [1], [5] y [12], se refiere a las siguientes publicaciones para un estudio en profundidad sobre los niveles sintácticos de los lenguajes de programación: [76], [77], [78] y [79].

no pongas las <sup>24</sup> ref. rama nota a pie

guienes capítulos del maual denominado EAB (expresiones aritmético booleanas)<sup>4</sup>. En este nuevo lenguaje vamos a ejemplificar como estos dos niveles sintácticos se relacionan entre sí agregando el operador *let*. Posteriormente explicaremos la relación de este operador con las variables y estudiaremos los conceptos de sustitución y  $\alpha$  – equivalencia

## 1 Sintáxis concreta

Definimos el lenguaje EAB (expresiones aritméticas-booleanas)<sup>5</sup> para ejemplificar los conceptos mencionados previamente de tal forma que la gramática para generar expresiones de lenguaje es de la siguiente forma:

**Definición 1.1.** Definición de la grámatica en forma BNF de EAB<sup>a</sup>:

$$\begin{aligned} E &::= T \mid E + T \\ T &::= F \mid T * F \\ F &::= N \mid (E) \\ N &::= D \mid ND \\ D &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

---

<sup>a</sup>Esta definición fue acuñada a partir de [1], [2], [5] y [12] adaptada para las instrucciones que nos interesa ejemplificar en este capítulo.

De forma equivalente, la gramática previamente mostrada tiene una representación en un conjunto de juicios lógicos.

Decimos que es entonces la sintáxis concreta de EAB definida de la siguiente forma:

**Definición 1.2.** Sintáxis concreta en forma de juicios lógicos para cada regla en nuestra gramática para EAB<sup>a</sup>:

$$\overline{0\ D} \quad \overline{1\ D} \quad \cdots \quad \overline{9\ D}$$

---

<sup>4</sup>Este lenguaje es una definición independiente y nueva a la que revisamos en el capítulo 1: Introducción, dado que iremos extendiendo y agregando nuevas instrucciones durante el desarrollo de los siguientes capítulos.

<sup>5</sup>Iniciaremos definiendo las instrucciones aritméticas primero y posteriormente agregaremos instrucciones lógico-booleanas como el operador *if*

$$\begin{array}{c}
 \frac{d D}{d N} \quad \frac{n N \quad d D}{nd N} \\
 \\ 
 \frac{n N}{n F} \quad \frac{e E}{(e) F} \\
 \\ 
 \frac{f F}{F T} \quad \frac{t T \quad f F}{t * f T} \\
 \\ 
 \frac{t T}{t E} \quad \frac{e E \quad t T}{e + t \quad E}
 \end{array}$$

<sup>a</sup>Esta definición fue acuñada a partir de [1], [2], [5] y [12] adaptada para las instrucciones que nos interesa ejemplificar en este capítulo.

## 2 Sintáxis abstracta

Como mencionamos al inicio de este capítulo, existe una representación en sintáxis abstracta para cada expresión bien formada de nuestro lenguaje EAB. Vamos a denotar como *"asa"* a dicha estructura, la cual está definida por el siguiente conjunto de reglas que nos permiten construir el árbol sintáctico correspondiente a dicha expresión.

**Definición 2.1.** Sintáxis Abstracta de EAB<sup>a</sup>

$$\frac{n \in \mathbb{N}}{num[n] \text{ asa}} \quad \frac{t_1 \text{ asa} \quad t_2 \text{ asa}}{sum(t_1, t_2) \text{ asa}} \quad \frac{t_1 \text{ asa} \quad t_2 \text{ asa}}{prod(t_1, t_2) \text{ asa}}$$

En donde el predicado  $t_1 \text{ asa}$  se lee como: "  $t_1$  es un árbol de sintáxis abstracta".

Es importante notar que los árboles de sintáxis abstracta aunque similares a la expresión en sintáxis concreta que representan corresponden a una categoría distinta de los niveles sintácticos para EAB.

<sup>a</sup>Esta definición fue acuñada a partir de [1], [2], [5] y [12] adaptada para las instrucciones que nos interesa ejemplificar en este capítulo.

### 2.1 Analizador sintáctico

Ahora bien, la relación  $e E \longleftrightarrow a \text{ asa}$  que enunciamos informalmente al inicio del capítulo, solo puede existir una vez que hayamos definido el analizador sintáctico que a su vez, define un algoritmo para transformar una expresión de sintáxis concreta a sintáxis abstracta expresado de la siguiente forma:

**Definición 2.2.** Analizador sintáctico para expresiones de lenguaje de EAB<sup>a</sup>

$$\frac{e \in \mathbb{N}}{e F \longleftrightarrow \text{num}[e] \text{ asa}}$$

$$\frac{e E \longleftrightarrow a \text{ asa}}{(e) F \longleftrightarrow a \text{ asa}}$$

$$\frac{e F \longleftrightarrow a \text{ asa}}{e T \longleftrightarrow a \text{ asa}}$$

$$\frac{e_1 T \longleftrightarrow a_1 \text{ asa} \quad e_2 F \longleftrightarrow a_2 \text{ asa}}{e_1 * e_2 T \longleftrightarrow \text{prod}(a_1, a_2) \text{ asa}}$$

$$\frac{e T \longleftrightarrow a \text{ asa}}{e E \longleftrightarrow a \text{ asa}}$$

$$\frac{e_1 E \longleftrightarrow a_1 \text{ asa} \quad e_2 T \longleftrightarrow a_2 \text{ asa}}{e_1 + e_2 T \longleftrightarrow \text{sum}(a_1, a_2) \text{ asa}}$$

<sup>a</sup>Definición formulada a partir de [1], [2], [5] y [12].

Ahora queremos extender EAB agregando instrucciones booleanas que permitan tener condicionales en nuestras expresiones de lenguaje. Es decir queremos agregar las constantes booleanas *True* y *False*, el operador *if ... then* simple y el operador *if... then ... else ...* así como parentizado de expresiones para evitar condiciones de "else colgante"<sup>6</sup> y ambigüedad (esta regla puede ser modelada forzando el paréntizado en la expresión inmediata a evaluar en la instrucción *then*).

**Ejercicio 2.1.** Define la sintaxis concreta para extender nuestro lenguaje EAB con las instrucciones lógicas previamente discutidas

$$\frac{\begin{array}{c} \text{True B} \quad \text{False B} \\ \hline \text{if } c \text{ then } t \text{ else } e E \end{array}}{\begin{array}{c} \frac{c B t \text{ LE } c E}{\text{if } c \text{ then } t \text{ E}} \quad \frac{x B}{x \text{ LE}} \\ \frac{e E}{(e) \text{ LE}} \quad \frac{e \text{ LE}}{e E} \end{array}}$$

tipos de  
fuerce

**Ejercicio 2.2.** Define la Sintaxis Abstracta para las nuevas expresiones lógicas

$$\frac{\begin{array}{c} \text{True B} \quad \text{False B} \\ \hline \text{T asa} \quad \text{F asa} \end{array}}{\begin{array}{c} \frac{c \text{ asa } t \text{ asa } c \text{ asa}}{\text{if}(c \text{ t }) \text{ asa}} \quad \frac{c \text{ asa } t \text{ asa}}{\text{if}(c \text{ t }) \text{ asa}} \end{array}}$$

**Ejercicio 2.3.** Define las reglas del analizador sintáctico para las nuevas expresiones lógicas.

$$\frac{\begin{array}{c} \text{True B} \longleftrightarrow \text{T asa} \quad \text{true} \quad \text{False B} \longleftrightarrow \text{F asa} \quad \text{false} \\ \hline c B \longleftrightarrow c' \text{ asa} \quad t \text{ LE } \longleftrightarrow t' \text{ asa} \quad e E \longleftrightarrow e' \text{ asa} \quad \text{if} \end{array}}{\begin{array}{c} \frac{\text{if } c \text{ then } t \text{ else } e E \longleftrightarrow \text{if}(c' \text{ t' } e') \text{ asa}}{} \end{array}}$$

<sup>6</sup>Este problema es conocido en inglés como "dangling else" recurrente en la implementación de *parsers* en el que una cláusula *else* opcional en una declaración *if-then(-else)* da como resultado que los condicionales anidados sean ambiguos[90].

$$\frac{\frac{c \text{ B} \longleftrightarrow c' \text{ asa} \quad t \text{ LE} \longleftrightarrow t' \text{ asa}}{\text{if}(c \text{ then } t) \text{ E} \longleftrightarrow \text{if}(c' t') \text{ asa}} \ if_s}{(e) \text{ LE} \longleftrightarrow e' \text{ asa} \quad \text{parent} \quad \frac{c \text{ B} \longleftrightarrow c' \text{ asa}}{c \text{ LE} \longleftrightarrow c' \text{ asa}} \quad \text{bool} \quad \frac{e \text{ LE} \longleftrightarrow e' \text{ asa}}{e \text{ E} \longleftrightarrow e' \text{ asa}} \quad \text{expr}}$$

### 3 El operador let

Ahora necesitamos extender aún mas nuestro EAB para incluir expresiones de tipo:

let  $x = e_1$  in  $e_2$  end

*tipo de letras*

Donde las apariciones de la variable  $x$  serán sustituidas por  $e_1$  en  $e_2$ .

Para ésto se tiene que definir la sintáxis de la nueva instrucción, el ligado de variables, y el operador *let*.

**Ejercicio 3.1.** Proporciona las sintáxis concreta para agregar las intrucciones *let* de tal forma que sea congruente con las especificación mencionada anteriormente.

$$\frac{e \text{ identificador}}{e \text{ V}} \quad \text{var} \quad \frac{e \text{ V}}{e \text{ Bool}} \quad vIn \quad \frac{e \text{ V}}{e \text{ F}} \quad vfac \quad \frac{x \text{ V} \quad e_1 \text{ E} \quad e_2 \text{ E}}{\text{let } x = e_1 \text{ in } e_2 \text{ end E}} \quad \text{let}$$

La sintáxis abstracta de orden superior define un tratamiento especial para las variables en donde éstas así como su alcance y ligado forman parte del metalenguaje con el que se define el árbol de sintáxis abstracta. Para ésto se agrega el constructor  $x.t$  que indica que la variable  $x$  está ligada en el árbol  $t$ , a este constructor se le llama abstracción<sup>7</sup>. Ésto es importante dado que al introducir las variables ligadas y su alcance tenemos que delimitar la expresión en la que la sustitución ocurrirá.

A continuación tenemos tres ejemplos que nos van a ayudar a ilustrar los diferentes niveles de EAB en donde tendremos que agregar juicios y reglas para poder utilizar esta instrucción<sup>8</sup>.

**Ejercicio 3.2.** Proporciona las Sintáxis Abstracta para agregar las instrucción *let* considerando el constructo  $x.t$ .

$$\frac{x \text{ identificador}}{\text{var}[x] \text{ asa}} \quad \frac{a_1 \text{ asa} \quad a_2 \text{ asa}}{\text{let}(x.a_1, a_2) \text{ asa}}$$

**Ejercicio 3.3.** Define las reglas de Análisis Sintáctico para agregar las instrucciones *let*

<sup>7</sup>Definición acuñada a partir de [1], [2], [5], [12], [80] y [81]

<sup>8</sup>Las definiciones de la sintáxis concreta, abstracta y el analizador sintáctico que están plasmadas en el ejercicio 3.1, 3.2 y 3.3 para el operador *let* fueron formuladas a partir de [2], [5], [12] y [81]

y las variables.

$$\frac{x \text{ identificador}}{x V \longleftrightarrow \text{var}[x] \text{ asa}} \quad \text{vara} \quad \frac{x V \quad e_1 E \longleftrightarrow a_1 \text{ asa} \quad e_2 E \longleftrightarrow a_2 \text{ asa}}{\text{let } x = e_1 \text{ in } e_2 \text{ end } E \longleftrightarrow \text{let}(e_1 \ x.e_2) \text{ asa}} \quad \text{leta}$$

**Ejercicio 3.4.** Obtén la representación en sintáxis abstracta partiendo de la sintáxis concreta de las siguiente expresión:

let  $x = 1$  in  $x + 1$  end

let(num[1],  $x.x + 1$ )

**Ejercicio 3.5.** Obtén la representación en sintáxis abstracta partiendo de la sintáxis concreta de las siguiente expresión:

let  $y = \text{if } True \text{ then } 1 \text{ else } 0$  in if  $x$  then  $False$  end

let(if(T, num[1], num[0]),  $y.\text{if}(\text{var}[x], F)$ )

**Ejercicio 3.6.** Obtén la representación en sintáxis abstracta partiendo de la sintáxis concreta de las siguiente expresión:

let  $x = False$  in if  $x$  then (if  $\alpha$  then (if  $\alpha$  then 4)) else  $False$  end

let(F,  $x.\text{if}(\text{var}[x], \text{if}(A, \text{if}(A, \text{num}[4]), F))$ )

### 3.1 Clasificación de variables en el operador let

La introducción de nuestro nuevo operador let acarrea consigo la definición de variable de ligado, variable ligada, variable libre y la definición de la función de sustitución y la función para obtener variables libres de una expresión, mismas que serán mostradas a continuación y serán de utilidad para resolver el resto de los ejercicios presentes en este capítulo.

**Definición 3.1** (Variable de ligado). La instancia de ligado de una variable es aquella que da a ésta su valor<sup>a</sup>. Es decir, es la aparición de la variable en donde esta es definida.

De tal forma que si tenemos la siguiente expresión:

$e = \text{let } x = e_1 \text{ in } e_2 \text{ end}$  una expresión de EAB

Decimos que  $x = e_1$  es la variable de ligado en nuestra expresión let.

<sup>a</sup>Definición formulada de [2], [5], [12], [80] y [81].

**Definición 3.2** (Variable ligada). Una variable está ligada si se encuentra contenida dentro del alcance de una variable de ligado con su nombre<sup>a</sup>.

De tal forma que si tenemos la siguiente expresión:

$$e = \text{let } x = e_1 \text{ in } e_2 \text{ end una expresión de EAB}$$

Decimos que  $x$  es la variable ligada en  $e_2$  dentro de nuestra expresión let.

---

<sup>a</sup>Definición formulada de [2], [5], [12], [80] y [81].

**Definición 3.3** (Variable libre). Una variable está libre en una expresión, si no se encuentra dentro del alcance de una variable de ligado con su nombre. Es decir, una variable es libre si no está ligada<sup>a</sup>.

De tal forma que si tenemos la siguiente expresión:

$$e = \text{let } x = e_1 \text{ in } y \text{ end una expresión de EAB}$$

Decimos que  $y$  es una variable libre en nuestra expresión let.

---

<sup>a</sup>Definición formulada de [2], [5], [12], [80] y [81].

**Definición 3.4** (Variables libres de una expresión). Dada una expresión en sintaxis abstracta  $a$  definimos el conjunto de variables libres de la expresión<sup>a</sup>, denotado  $FV(a)$ , como sigue:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(O(a_1, \dots, a_n)) &= FV(a_1) \cup \dots \cup FV(a_n) \\ FV(x.a) &= FV(a) \setminus \{x\} \end{aligned}$$

---

<sup>a</sup>Definición extraída de [2], [5], [12], [80] y [81].

**Ejercicio 3.7.** Dada la siguiente expresión let definida como:

$$\begin{aligned} e =_{\text{def}} &\text{let } x = \text{let } x = (2 + 4) \text{ in } x + 1 \text{ end in} \\ &\text{let } y = (x * 2) + x \text{ in} \\ &\quad (x * y) + \text{let } z = 7 \text{ in } y + z \text{ end} \\ &\quad \text{end} \\ &\text{end} \end{aligned}$$

Por cada subexpresión let en  $e$  (inclusive) proporciona el alcance indicado o subrayando las variables ligadas.

```

 $e =_{\text{def}} \text{let } \underline{x} = \text{let } \underline{x} = (2 + 4) \text{ in } \underline{x} + 1 \text{ end in}$ 
 $\quad \text{let } \underline{y} = (\underline{x} * 2) + \underline{x} \text{ in}$ 
 $\quad \quad (\underline{x} * \underline{y}) + \text{let } \underline{z} = 7 \text{ in } \underline{y} + \underline{z} \text{ end}$ 
 $\quad \text{end}$ 
 $\text{end}$ 

```

**Ejercicio 3.8.** Escribe cada subexpresión let en  $e$  (inclusive) indicando la variable de ligado y la expresión let para la cual la variable de ligado tiene alcance.

```

 $\text{let } \underline{x} = \text{let } x = (2 + 4) \text{ in } x + 1 \text{ end in}$ 
 $\quad \text{let } y = (x * 2) + x \text{ in}$ 
 $\quad \quad (x * y) + \text{let } z = 7 \text{ in } y + z \text{ end}$ 
 $\quad \text{end}$ 
 $\text{end}$ 

```

La siguiente expresión let es la que le da su valor a la variable más externa definida en  $e$  la cuál se identifica como:

```
 $\text{let } \underline{x} = (2 + 4) \text{ in } x + 1 \text{ end}$ 
```

La expresión let que continua está delimitada por los marcadores más externos in end en  $e$  y se identifica como:

```

 $\text{let } \underline{y} = (x * 2) + x \text{ in}$ 
 $\quad (x * y) + \text{let } z = 7 \text{ in } y + z \text{ end}$ 
 $\quad \text{end}$ 

```

La última expresión let está delimitada por los marcadores in end de la expresión anterior y se identifica como:

```
 $\text{let } \underline{z} = 7 \text{ in } y + z \text{ end}$ 
```

**Ejercicio 3.9.** La expresión  $e$  se puede evaluar a un valor? si sí muestra el procedimiento para la evaluación informal (aún no definimos la manera en la que se va a evaluar la expresiones del lenguaje EAB pero se puede dar una idea de la evaluación partiendo de la descripción hecha en el capítulo).

```


$$e =_{\text{def}} \text{let } x = \text{let } x = (2 + 4) \text{ in } x + 1 \text{ end in}$$


$$\quad \text{let } y = (x * 2) + x \text{ in}$$


$$\quad \quad (x * y) + \text{let } z = 7 \text{ in } y + z \text{ end}$$


$$\quad \text{end}$$


$$\text{end}$$


```

Primero se resuelve la expresión let que la da valor a la variable  $x$  en la expresión  $e$ :

$$\text{let } x = (2 + 4) \text{ in } x + 1 \text{ end} = (2 + 4) + 1 = (6) + 1 = 7$$

Se sustituye este valor en el cuerpo de  $e$  resultando en la siguiente expresión:

```


$$\text{let } y = (7 * 2) + 7 \text{ in}$$


$$\quad (7 * y) + \text{let } z = 7 \text{ in } y + z \text{ end}$$


$$\quad \text{end}$$


```

Resolviendo la operación para dar variable a la variable  $y$  obtenemos el siguiente resultado:

$$y = (7 * 2) + 7 = (14) + 7 = 21$$

Sustituyendo el nuevo valor en el cuerpo de la expresión anterior se obtiene:

$$= (7 * 21) + \text{let } z = 7 \text{ in } 21 + z \text{ end}$$

Resolviendo las operaciones y la sustitución de la variable  $z$  en el cuerpo de la expresión let se obtiene:

$$= 147 + 21 + 7 = 168 + 7 = 175$$

**Ejercicio 3.10.** En nuestra expresión  $e$  hay variables libres, argumenta por qué si o por qué no.

No, todas las variables definidas en la expresión  $e$  están asociadas a un operador let marcando su alcance en las subexpresiones presentes por lo que ninguna variable aparece sin su respectiva asociación.

## 4 Sustitución y $\alpha$ -equivalencias

Para finalizar este capítulo se introduce el concepto de alfa equivalencia:

**Definición 4.1** ( $\alpha$ -equivalencia). Se dice que dos expresiones  $e_1$  y  $e_2$  son  $\alpha$ -equivalentes si y sólo si difieren únicamente en el nombre de las variables ligadas. Y se escribe  $e_1 \equiv_{\alpha} e_2$ <sup>a</sup>.

---

<sup>a</sup>La notación de la relación de  $\alpha$ -equivalencia fue extraída de [1], [5] y [12], la definición formulada a partir de [81] y [89]

Esta definición resulta de utilidad para trabajar con expresiones que necesiten ser removidas a lo más en sus variables ligadas para poder aplicar la sustitución sobre dicha expresión cuando haya un conflicto ocasionado por los identificadores de las variables.

A continuación definimos la función de sustitución aplicada a expresiones pertenecientes nuestro lenguaje EAB las expresiones let, operaciones lógicas if.

**Definición 4.2** (Función de Sustitución). Se define la función de sustitución<sup>a</sup> sobre los árboles de sintaxis abstracta de orden superior como sigue:

$$\begin{aligned}
 x[x := e] &= e \\
 z[x := e] &= z \\
 num[n][x := e] &= num[n] \\
 sum(a_1, a_2)[x := e] &= sum(a_1[x := e], a_2[x := e]) \\
 prod(a_1, a_2)[x := e] &= prod(a_1[x := e], a_2[x := e]) \\
 let(a_1, a_2)[x := e] &= let(a_1[x := e], a_2[x := e]) \\
 if(a_1, a_2, a_3)[x := e] &= if(a_1[x := e], a_2[x := e], a_3[x := e]) \\
 (z.a)[x := e] &= z.(a[x := e]) && \text{Si } x \neq z \text{ y } z \notin FV(e) \\
 (z.a)[x := e] &= \text{indefinido} && \text{Si } z \in FV(e)
 \end{aligned}$$

---

<sup>a</sup>Esta definición está formulada a partir de [1], [2], [5] y [12] adaptada para las instrucciones que hemos revisado hasta este momento para EAB.

**Ejercicio 4.1.** Proporciona una expresión  $\alpha$ -equivalente a la siguiente expresión.

```

let x = let y = False in if (y then 1 else 0) in (x + 2) * w end

let k = let j = False in if (j then 1 else 0) in (k + 2) * w end
  
```

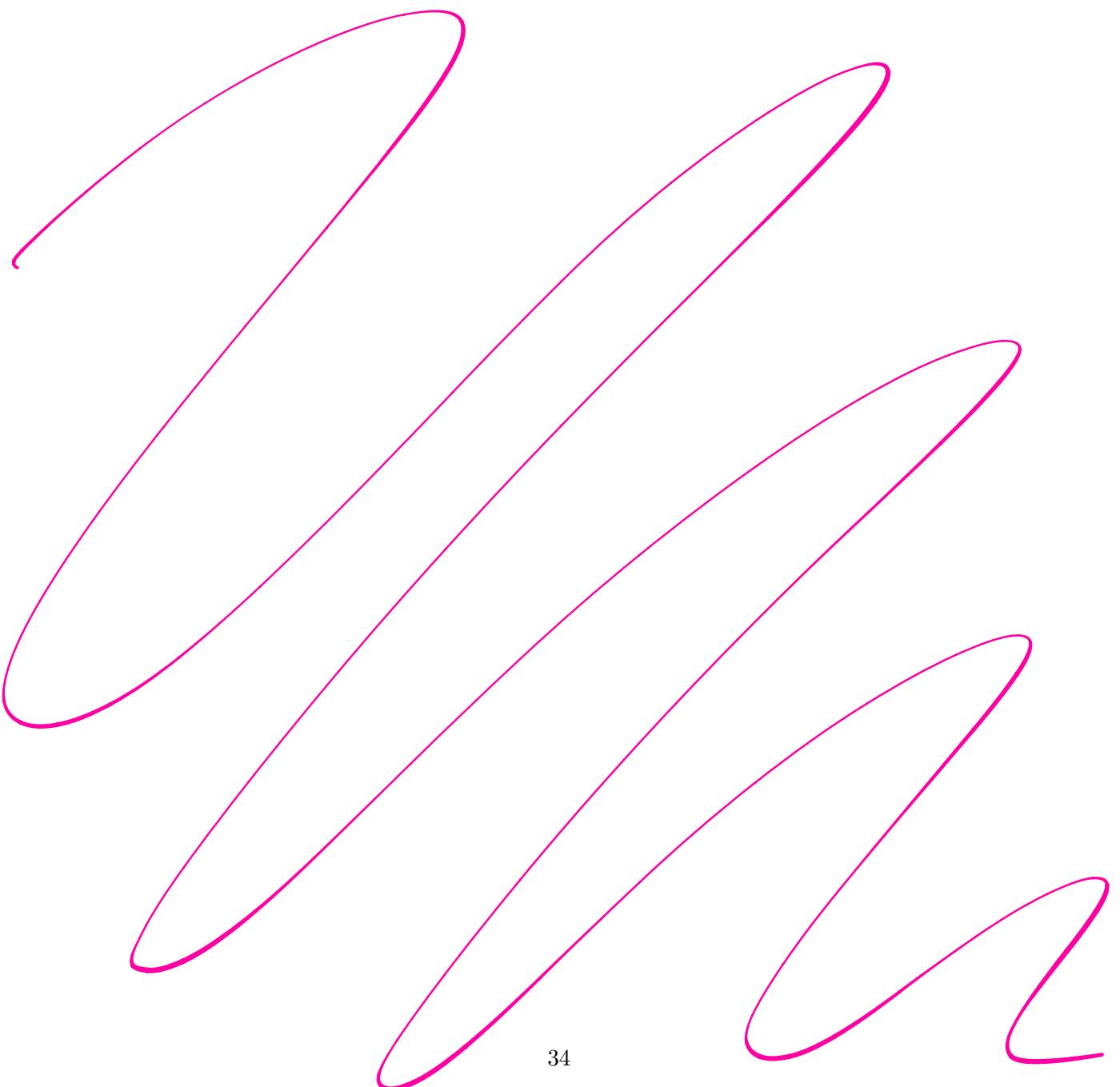
**Ejercicio 4.2.** Proporciona una expresión  $\alpha$ -equivalente a la expresión  $e$  definida en la sección anterior.

```

e\alpha = let a = let d = (2 + 4) in d + 1 end in
      let b = (a * 2) + a in
          (a * b) + let c = 7 in b + c end
      end
  end
  
```

**Ejercicio 4.3.** Desarrolla la sustitución aplicada a la siguiente expresión de acuerdo con la función de sustitución previamente definida.

```
let x = (w + 1) * 2 in if ( True then w else (2 * x)) end [w := 2]  
= let x = ((w + 1) * 2)[w := 2] in if ( True then w else (2 * x))[w := 2] end  
= let x = ((w+1)[w := 2]*2[w := 2]) in if ( True [w := 2] then w[w := 2] else (2*x)[w := 2]) end  
= let x = ((w[w := 2]+1[w := 2])*2) in if ( True then 2 else (2[w := 2]*x[w := 2])) end  
= let x = ((2 + 1) * 2) in if ( True then 2 else (2 * x)) end
```



## 5 Ejercicios para el lector

**Ejercicio 5.1.** Explica con tus palabras la diferencia entre la sintaxis abstracta y la sintaxis concreta.

**Ejercicio 5.2.** Explica con tus palabras por que es necesaria la sintaxis abstracta de orden superior para modelar el alcance de una variable con el constructor `x.t` ?

**Ejercicio 5.3.** Proporciona el árbol de sintaxis abstracta para la siguiente expresión `let`

`let z = 777 in (w + h) * (z) + let v = -33 in (z * z) + v end end`

**Ejercicio 5.4.** Proporciona el árbol de sintaxis abstracta para la siguiente expresión `if`

`if ( True then (if False then 0) else (2 * 7) - 1 )`

**Ejercicio 5.5.** La siguiente expresión es una expresión correcta de nuestro lenguaje? Explica por que si o porque no de acuerdo a las reglas de sintaxis concreta definidas.

*Demostrar* `if ( ⊤ then let x = True in if ( x then ⊥ ) end )`

**Ejercicio 5.6.** Dada la siguiente expresión `let` definida como:

```
e1 =def let x = let x = 2 in let y = 3 in y + x end end in  
      let y = (x * 2) * 19 in  
        (x + y) * let z = let v = 8 in z end in (y + z) * v end  
          end  
            end
```

Proporciona lo siguiente por cada subexpresión `let` definida en `e1`:

- A) Las variables ligadas
- B) Las variables libres
- C) El alcance de cada variable
- D) El valor que se obtiene al desarrollar la expresión.

**Ejercicio 5.7.** Proporciona una expresión  $\alpha$ -equivalente a la `e1` del ejercicio anterior

**Ejercicio 5.8.** Desarrolla la siguiente sustitución, en caso de ser inválida explica por qué.

`( let z = 3 in (w + z) + if( True then 45 else y ) end ) [y := 7]`

**Ejercicio 5.9.** Desarrolla la siguiente sustitución paso por paso, en caso de ser inválida explica por qué.

let  $z = 3$  in  $(w + z) + \text{if}(\text{True} \text{ then } 45 \text{ else } y) \text{ end}$  } [y := z + 7]

**Ejercicio 5.10.** A continuación se define la sintáxis concreta para un lenguaje funcional simple:

$$e ::= x \parallel n \parallel e_1 e_2 \parallel \text{fun}(x) \rightarrow e$$

En donde el primer constructor representa las variables del lenguaje, el segundo números naturales, el tercero aplicación de función y el último la definición de funciones.

De acuerdo a la especificación discutida anteriormente, contesta los siguientes incisos

- enunciado*
- Traduce la gramática anterior a una definición inductiva con reglas de inferencia.
  - Diseña una sintáxis abstracta apropiada para este lenguaje (Primero observa si es necesario alguna especie de ligado como el que define el operando **let** discutido en este capítulo).
  - Escribe las reglas para la relación de análisis sintáctico ( $\longleftrightarrow$ ) del lenguaje.
  - Diseña un algoritmo de sustitución para este lenguaje.
  - La relación de  $\alpha$ -equivalencia en este lenguaje se da respecto al operador de definición de función fun, se dice que dos expresiones son  $\alpha$ -equivalentes si solo difieren en el nombre de la variable del parámetro de la función. Por ejemplo:

$$\text{fun}(x) \rightarrow e \equiv_{\alpha} \text{fun}(y) \rightarrow e$$

Demuestra que  $\equiv_{\alpha}$  es una relación de equivalencia.

## Capítulo 4

# Semántica



En el capítulo anterior se estudió la composición sintáctica de las expresiones de nuestro lenguaje mediante juicios que nos permiten construir la expresión concreta o bien el árbol de sintáxis abstracta asociado a la misma. Esta estructura nos permite responder a la pregunta: ”¿Qué es una expresión de lenguaje?”

AMP

En el presente capítulo nos ocuparemos de la semántica del lenguaje, que tiene como propósito responder a la pregunta: ”¿Qué significa una expresión de lenguaje?”. En este curso la respuesta será: ”El comportamiento que tiene la expresión del lenguaje en tiempo de ejecución al ser evaluada”. En particular nos interesa obtener el valor al que la expresión del lenguaje se reduce (semántica dinámica) y en ocasiones será de interés analizar las expresiones del lenguaje y que su estructura sea sintácticamente correcta (semántica estática).

Muchas veces en la documentación oficial de los lenguajes de programación la semántica está escrita en un manual el cual contiene una descripción de alto nivel acerca de como una determinada instrucción, método o expresión se comporta al momento de ejecutarse. Si bien ésto resulta útil al escribir un programa no es el enfoque de este curso.

# Section o Subsection

## Objetivo

En este capítulo nuestro principal interés será el de definir la semántica de **EAB** en sus dos niveles: estático y dinámico<sup>1</sup>. cada nivel contará con sus propias reglas de inferencia que dictaminan el proceso que se debe seguir al para evaluar o asignar un tipo a una expresión bien formada.

## Planteamiento

Iniciaremos el estudio de este capítulo presentando las reglas para poder asignar un tipo a cada un de las expresiones bien formadas de EAB, Este nivel corresponde a la sémantica estática y servirá como un primer paso para aportar seguridad al lenguaje y garantizar que se pueda regresar un valor al final de la evaluación de dicha expresión.

Posteriormente se definirá el procedimiento para evaluar las expresiones bien formadas de **EAB**. Este mecanismo se conoce como semántica dinámica y se estudiará en sus dos paradigmas: de paso grande y de paso pequeño.

Por último definiremos la función **eval** para **EAB**.

## 1 Semántica estática

La semántica estática extrae información del programa en tiempo de compilación, la cantidad y calidad de la información obtenida depende de la implementación puntual de cada compilador. Información como el alcance de una variable, el tipado de una expresión y en particular saber si una expresión tiene variables libres serán de utilidad en los siguientes capítulos para evaluar las expresiones de nuestro lenguaje.

Éste último punto es importante ya que necesitamos que todas las variables estén ligadas y no haya presencias libres en nuestras expresiones, de lo contrario de la evaluación no se obtendrá valor alguno.

**Definición 1.1** (Semántica estática para capturar expresiones con variables libres en EAB). En el lenguaje de expresiones aritméticas con el que hemos estado trabajando, las expresiones de la forma:

```
let x = y in x + x end
```

son sintácticamente correctas pero semánticamente incorrectas, pues la variable y está libre en la expresión por lo que no podría evaluarse.

Ésta es una de las propiedades de las que se encarga la semántica estática, así que daremos un conjunto de reglas para definir una semántica estática encargada de evitar

<sup>1</sup>Las definiciones presentadas en este capítulo fueron extraídas de: Enríquez Mendoza J., Lenguajes de Programación Notas de Clase: Semántica. Univerisadad Nacional Autónoma de México. 2022.

estos errores. Para esto se define un juicio

$$\Delta \sim e$$

En donde  $\Delta$  es un conjunto de variables en donde se guardan las variables previamente definidas y  $e$  una expresión del lenguaje en sintaxis abstracta, y se lee como  $e$  no tiene variables libres bajo el conjunto  $\Delta$  de variables definidas. Entonces se definen las reglas de semántica estática como:

$$\frac{x \in \Delta}{\Delta \sim x} \text{ fvv} \quad \frac{}{\Delta \sim [n]} \text{ fvn} \quad \frac{\Delta, x \sim e}{\Delta \sim x.e} \text{ fva}$$

$$\frac{}{\Delta \sim [Bool]} \text{ fvb} \quad \frac{\Delta \sim e_1 \cdots \Delta \sim e_n}{\Delta \sim O(e_1, \dots, e_n)} \text{ fvo}$$

Para garantizar que una expresión  $e$  no tiene variables libres se inicia con el conjunto vacío de variables definidas y se debe probar  $\emptyset \sim e$  usando las reglas anteriores.

**Definición 1.2** (Expresión cerrada). Se dice que una expresión  $e$  del lenguaje es cerrada si no tiene apariciones de variables libres, es decir,  $e$  es cerrada si y sólo si  $\emptyset \sim e$ .

**Ejercicio 1.1.** Para la siguiente expresión realiza el análisis estático para encontrar variables libres mediante la derivación aplicando las reglas del juicio  $\sim$

$$\begin{array}{c} \text{let } x = 1 \text{ in } (y * x) + z \text{ end} \\ \\ \frac{\frac{\frac{\frac{\frac{\text{Error}}{\{x\} \sim y}}{\frac{x \in \{x\}}{\{x\} \sim x}} \text{ fvn}}{\{x\} \sim (y * x)} \text{ fvo}}{\{x\} \sim (y * x) + z} \text{ fvo}}{\emptyset \sim \text{let } x = 1 \text{ in } (y * x) + z \text{ end}} \text{ fva} \end{array}$$

**Ejercicio 1.3.** Para la siguiente expresión realiza el análisis estático para encontrar variables libres mediante la derivación aplicando las reglas del juicio  $\sim$

$$\begin{array}{c} 1 + \text{if} ( \text{ let } x = \text{True} \text{ in } x \text{ end} \text{ then } 2 \text{ else } 1 ) \\ \\ \frac{\frac{\frac{\frac{\frac{\frac{\text{fvn}}{\{x\} \sim x}}{\Delta \sim \text{let } x = \text{True} \text{ in } x \text{ end}} \text{ fvo}}{\Delta \sim \text{if} ( \text{ let } x = \text{True} \text{ in } x \text{ end} \text{ then } 2 \text{ else } 1 )} \text{ fvn}}{\Delta \sim 2} \text{ fvn}}{\Delta \sim 1} \text{ fvn}}{\Delta \sim 1 + \text{if} ( \text{ let } x = \text{True} \text{ in } x \text{ end} \text{ then } 2 \text{ else } 1 )} \text{ fvo} \end{array}$$

Hay que agregar los corchetes  
de Gabarito de S.E.

De los ejercicios anteriores podemos notar que el análisis sintáctico para evaluar expresiones con variables libres fallará en una o más ramas, mientras que los árboles de derivación<sup>2</sup>, aplicados en expresiones cerradas concluirán todas sus ramas con algún axioma (**fvn**, **fvb**).

Hay que evitar usar bold

## 2 Semántica dinámica

La semántica dinámica que estudiaremos en este capítulo, es el siguiente componente del proceso de ejecución de un programa. Una vez verificada la estructura dada por la semántica estática se puede comenzar a discutir el **cómo** se modelará la ejecución de las expresiones correctas del lenguaje.

Diferentes modelos de ejecución pueden ser aplicados dependiendo del enfoque que se quiera tener para estudiar ésta (cambios en la memoria, el valor las variables del programa, el valor final al que se evalúa la expresión, etc). En este capítulo nos centraremos en la **Semántica Operacional**, que estudia el proceso de ejecución modelando cada configuración del programa como un estado (una expresión del lenguaje) y las transiciones entre ellos que serán los cálculos que partiendo de una configuración A nos permiten obtener una configuración B.

### 2.1 Semántica operacional

En esta categoría se hace la distinción entre dos tipos de semántica

- **Estructural** ó también conocida de paso pequeño, la cual modela la ejecución de un programa describiendo las transiciones una a una mostrando los cálculos generados de forma individual.
- **Natural** ó también conocida de paso grande, la cual modela de forma general la ejecución de un programa que fue llevada a cabo para obtener el resultado.

Como ambos enfoques solo difieren en el sistema de transición que emplean, definiremos los estados de la máquina de transición para ambos a continuación.

**Definición 2.1** (Sistema de transición para semántica operacional de EAB). Se define la semántica operacional del lenguaje de expresiones aritméticas utilizando el sistema de transición siguiente:

**Conjunto de estados**  $S = \{a \mid a\}$ , es decir, los estados del sistema son las expresiones bien formadas del lenguaje en sintaxis abstracta. Esta definición corresponde a

<sup>2</sup>Este tipo de árboles de derivación han sido discutidos antes en el cap. 2 **Herramientas Matemáticas**

la forma de modelar el proceso de ejecución por un

la regla de inferencia:

$$\frac{a \text{ asa}}{a \text{ estado}} \text{ state}$$

**Estados Iniciales**  $I = \{a \mid a, \emptyset \sim a\}$ , los estados iniciales son todas las expresiones cerradas del lenguaje, es decir, expresiones sin variables libres. Correspondiente a la regla:

$$\frac{a \text{ asa} \quad \emptyset \sim a}{a \text{ inicial}} \text{ init}$$

**Estados Finales** se definen como las expresiones que representan a los posibles resultados finales de un proceso de evaluación. Para poder modelarlos definimos una categoría de valores los cuales son un subconjunto de expresiones que ya se han terminado de evaluar y no pueden reducirse más, con el juicio  $v$ . Para el caso de el único valor son los números, formalmente definido con la regla:

$$\frac{}{num[n] \text{ valor}} \text{ vnum} \quad \frac{}{bool[b] \text{ valor}} \text{ vbool}$$

Entonces se define el conjunto de estados finales  $F = \{a \mid a \text{ valor}\}$ , correspondiente a la regla:

$$\frac{a \text{ valor}}{a \text{ final}} \text{ fin}$$

De la definición anterior es importante recalcar que los estados finales asociados a un valor no pueden ser reducidos a ningún otro estado, es por ésto que los denotaremos como estados "bloqueados"

**Definición 2.2** (Estado bloqueado). Un estado  $s$  está bloqueado si no existe otro estado  $s'$  tal que  $s \rightarrow s'$  y lo denotamos como  $s \not\rightarrow$ .

## 2.2 Semántica de paso pequeño

Para esta semántica las transiciones se modelarán paso a paso mediante la función de transición, denotada como:  $e_1 \rightarrow e_2$  donde  $e_1$  es llamado "*reducto*" y  $e_2$  es llamado "*reducto.en*" donde se interpreta cómo la transición entre  $e_1$  y  $e_2$  si y solo si en un paso de evaluación se puede reducir  $e_1$  a  $e_2$ .

La siguiente definición contiene las reglas para cada constructor de nuestro lenguaje **EAB** y las posibles reducciones para transitar de un estado a otro.

**Definición 2.3** (Función de transición para semántica de paso pequeño). Se da la definición de la función de transición para completar la definición de la semántica operacional de paso pequeño con el sistema de transición 2.1 mediante las siguientes reglas de inferencia:

# hipergrafos

## Suma

$$\frac{\frac{a_1 \rightarrow a'_1}{\text{suma}(a_1, a_2) \rightarrow \text{suma}(a'_1, a_2)} \text{ suma1} \quad \frac{a_2 \rightarrow a'_2}{\text{suma}(num[n], a_2) \rightarrow \text{suma}(num[n], a'_2)} \text{ suma2}}{\text{suma}(num[n], num[m]) \rightarrow num[n +_{\mathbb{N}} m]} \text{ sumaf}$$

## Producto

$$\frac{\frac{a_1 \rightarrow a'_1}{\text{prod}(a_1, a_2) \rightarrow \text{prod}(a'_1, a_2)} \text{ prod1} \quad \frac{a_2 \rightarrow a'_2}{\text{prod}(num[n], a_2) \rightarrow \text{prod}(num[n], a'_2)} \text{ prod2}}{\text{prod}(num[n], num[m]) \rightarrow num[n \times_{\mathbb{N}} m]} \text{ prod}$$

## Expresiones lógicas

$$\frac{\frac{\frac{if(\text{True}, e_1, e_2) \rightarrow e_1}{if(\text{False}, e_1, e_2) \rightarrow e_2} \text{ ift} \quad \frac{a_1 \rightarrow a'_1}{if(a_1, e_1, e_2) \rightarrow if(a'_1, e_1, e_2)} \text{ if1}}{if(a_1, e_1, e_2)n \rightarrow if(a'_1, e_1, e_2)} \text{ ifn}}$$

## Asignaciones locales

$$\frac{\frac{v \text{ valor}}{let(v, x.a_2) \rightarrow a_2[x := v]} \text{ letf} \quad \frac{a_1 \rightarrow a'_1}{let(a_1, x.a_2) \rightarrow let(a'_1, x.a_2)} \text{ let1}}{let(v, x.a_2) \rightarrow a_2[x := v] \rightarrow let(a'_1, x.a_2)}$$

**Nota:** la evaluación del operador *if* es una evaluación "perezosa", en donde no se evaluará el resto de la expresión if hasta antes haber determinado el valor de la sentencia de control (**True** ó **False**)

**Ejercicio 2.1.** Dada la siguiente expresión de nuestro lenguaje **EAB** utiliza las reglas de transición para semántica de paso pequeño para evaluarla.

**let**  $k = (3 + 1)$  **in**  $(7 * k) + 1$  **end**

$$\begin{aligned}
 & \text{let}(\text{sum}(3, 1), k.\text{sum}(\text{prod}(7, k), 1)) \\
 \rightarrow & \text{let}(4, k.\text{sum}(\text{prod}(7, k), 1)) \\
 \rightarrow & (\text{sum}(\text{prod}(7, k), 1))[k := 4] \\
 \rightarrow & \text{sum}(\text{prod}(7, k)[k := 4], 1[k := 4]) \\
 \rightarrow & \text{sum}(\text{prod}(7[k := 4], k[k := 4]), 1) \\
 \rightarrow & \text{sum}(\text{prod}(7, 4), 1) \\
 \rightarrow & \text{sum}(28, 1) \\
 \rightarrow & 29
 \end{aligned}$$

**Ejercicio 2.2.** Dada la siguiente expresión de nuestro lenguaje **EAB** utiliza las reglas de transición para semántica de paso pequeño para evaluarla.

$$\text{if } (\text{False} \text{ then } 4 * \text{let } x = 99 * 99 \text{ in } x + x \text{ end else } 0)$$

$$\begin{aligned} & \text{if}(\text{False}, \text{let}(\text{prod}(99, 99)x.\text{sum}(x, x), 0) \\ & \rightarrow 0 \end{aligned}$$

Del ejercicio anterior podemos notar la ventaja de la evaluación perezosa, dado que nuestro valor de control es *False* no es necesario evaluar la expresión  $e_1$  y directamente nos saltaremos a evaluar  $e_2$  por la regla **iff** que en este caso es el valor 0.

**Ejercicio 2.3.** Dada la siguiente expresión de nuestro lenguaje **EAB** utiliza las reglas de transición para semántica de paso pequeño para evaluarla.

$$\text{let } x = \text{if } (\text{True} \text{ then } 42 \text{ else } 0) \text{ in } \text{if } (\text{False} \text{ then } 41 \text{ else } x)$$

$$\begin{aligned} & \text{let}(\text{if}(\text{True}, 42, 0), x.\text{if}(\text{False}, 41, x)) \\ & \rightarrow \text{let}(42, x.\text{if}(\text{False}, 41, x)) \\ & \rightarrow \text{if}(\text{False}, 41, x)[x := 42] \\ & \rightarrow \text{if}(\text{False}[x := 42], 41[x := 42], x[x := 42]) \\ & \rightarrow \text{if}(\text{False}, 41, 42) \\ & \rightarrow 42 \end{aligned}$$

Es importante definir las características inductivas que la relación de transición posee para alcanzar estados partiendo de uno inicial.

Éstas se conocen como **cerraduras** y pueden ser: **reflexiva** (un estado puede llegar a si mismo en 0 aplicaciones de pasos), **transitiva** (si un estado  $e_1$  puede alcanzar un estado  $e_2$  y  $e_2$  puede alcanzar a  $e_n$  en un número finito de pasos entonces  $e_1$  puede llegar a  $e_n$  en un número finito de pasos) ó **positiva** (la aplicación de las reglas de transición n veces con  $n \geq 1$ ).

A continuación enunciamos las cerraduras, la relación de transición y la iteración en n pasos.

**Definición 2.4** (Cerradura transitiva y reflexiva). La cerradura reflexiva y transitiva se denota como  $\rightarrow^*$  y se define con la siguientes reglas:

$$\frac{}{s \rightarrow^* s} \quad \frac{s_1 \rightarrow s_2 \quad s_2 \rightarrow^* s_3}{s_1 \rightarrow^* s_3}$$

Intuitivamente la relación  $s_1 \rightarrow^* s_2$  modela que es posible llegar desde  $s_1$  hasta  $s_2$  en un número finito de pasos (posiblemente 0), de la relación de transición  $\rightarrow$ .

**Definición 2.5** (Cerradura positiva). La cerradura transitiva se denota como  $\rightarrow^+$  y se define con la siguientes reglas:

$$\frac{s_1 \rightarrow s_2}{s_1 \rightarrow^+ s_2} \quad \frac{s_1 \rightarrow s_2 \quad s_2 \rightarrow^+ s_3}{s_1 \rightarrow^+ s_3}$$

Intuitivamente la relación  $s_1 \rightarrow^+ s_2$  modela que es posible llegar desde  $s_1$  hasta  $s_2$  en un número finito de pasos estrictamente mayor a cero, de la relación de transición  $\rightarrow$ . Es decir, se llega de  $s_1$  a  $s_2$  en al menos un paso.

**Definición 2.6** (Iteración en  $n$  pasos). la iteración en  $n$  pasos se denota como  $\rightarrow^n$  con  $n \in \mathbb{N}$  y se define con la siguientes reglas:

$$\frac{}{s \rightarrow^0 s} \quad \frac{s_1 \rightarrow s_2 \quad s_2 \rightarrow^n s_3}{s_1 \rightarrow^{n+1} s_3}$$

Intuitivamente la relación  $s_1 \rightarrow^n s_2$  modela que es posible llegar desde  $s_1$  hasta  $s_2$  en exactamente  $n$  pasos de la relación de transición  $\rightarrow$ .

## 2.3 Semántica de paso grande

Este semántica define la ejecución de un programa mostrando el valor al cual se evalúa. Encapsula de forma general el proceso sin detallar paso a paso como es que una expresión es evaluada (contrario a la semántica de paso pequeño).

La relación de transición en este caso se denota como  $e \Downarrow v$  donde  $e$  es una expresión válida de nuestro lenguaje **EAB** y  $v$  es un valor, ésta se lee como "la expresión  $e$  se evalúa a  $v$ ".

**Definición 2.7** (La relación  $\Downarrow$  para ). Se define la transición sobre los estados definidos en 2.1 para la semántica operacional de paso grande de mediante las siguientes reglas de inferencia:

$$\begin{array}{c} \frac{}{num[n] \Downarrow num[n]} \text{ bsnum} \quad \frac{}{bool[b] \Downarrow bool[b]} \text{ bsbool} \\ \frac{e_1 \Downarrow [n] \quad e_2 \Downarrow [m]}{sum(e_1, e_2) \Downarrow [n +_{\mathbb{N}} m]} \text{ bssum} \\ \frac{e_1 \Downarrow [n] \quad e_2 \Downarrow [m]}{prod(e_1, e_2) \Downarrow [n \times_{\mathbb{N}} m]} \text{ bsprod} \\ \frac{e_1 \Downarrow v_1 \quad e_2[x := v_1] \Downarrow v_2}{let(e_1, x.e_2) \Downarrow v_2} \text{ bslet} \\ \frac{e_0 \Downarrow True \quad e_1 \Downarrow v_1}{if(e_0, e_1, e_2) \Downarrow v_1} \text{ bsift} \quad \frac{e_0 \Downarrow False \quad e_2 \Downarrow v_2}{if(e_0, e_1, e_2) \Downarrow v_2} \text{ bsift} \end{array}$$

**Nota:** Para este enfoque si es necesario definir una regla de transición para los valores del lenguaje, pues es necesaria para el correcto funcionamiento del resto de las reglas.

**Teorema 2.8** (Equivalencia entre semántica de paso pequeño y paso grande). Para cualquier expresión  $e$  del lenguaje **EAB** se cumple:

$$e \rightarrow^* v \text{ si y sólo si } e \Downarrow v$$

Es decir, las semánticas que hemos definido son equivalentes.

**Ejercicio 2.4.** Dada la siguiente expresión de **EAB** evalúala utilizando **semántica de paso grande**. Adicionalmente proporciona la representación en sintaxis abstracta.

Sintaxis Concreta:

`let x = let y = False in if ( y then 0 else 1 ) end in x + 1 end`

Sintaxis Abstracta:

`let(let(False, y.if(y, 0, 1)), x.sum(x, 1))`

Evaluación Paso Grande:

$$\frac{\frac{\frac{False \Downarrow False}{\text{bsbool}} \quad \frac{\text{if}(y, 0, 1)[y := False] \Downarrow 1}{\text{bslet}}}{\text{let}(False, y.\text{if}(y, 0, 1)) \Downarrow 1} \quad \frac{\text{sum}(x, 1)[x := 1] \Downarrow 2}{\text{bssum}}}{\text{bslet}}}{\text{let}(\text{let}(False, y.\text{if}(y, 0, 1)), x.\text{sum}(x, 1)) \Downarrow 2}$$

**Ejercicio 2.5.** Dada la siguiente expresión de **EAB** evalúala utilizando **semántica de paso grande**. Adicionalmente proporciona la representación en sintaxis abstracta.

Sintaxis Concreta:

$((7 + 4) * 4) + ((8 + 3) * 2)$

Sintaxis Abstracta

`sum(prod(sum(7, 4), 4), prod(sum(8, 3), 2))`

Evaluación Paso Grande:

$$\frac{\frac{\frac{\frac{7 \Downarrow 7}{\text{bsnum}} \quad \frac{4 \Downarrow 4}{\text{bsnum}}}{\text{bsum}}}{\text{sum}(7, 4) \Downarrow 11} \quad \frac{\frac{4 \Downarrow 4}{\text{bsnum}}}{\text{bsum}}}{\text{bsprod}} \quad \frac{\frac{\frac{8 \Downarrow 8}{\text{bsnum}} \quad \frac{3 \Downarrow 3}{\text{bsnum}}}{\text{bsum}}}{\text{sum}(8, 3) \Downarrow 11} \quad \frac{\frac{2 \Downarrow 2}{\text{bsnum}}}{\text{bsprod}}}{\text{bssum}}$$

$$\frac{\text{prod}(\text{sum}(7, 4), 4) \Downarrow 44 \quad \text{prod}(\text{sum}(8, 3), 2) \Downarrow 22}{\text{sum}(\text{prod}(\text{sum}(7, 4), 4), \text{prod}(\text{sum}(8, 3), 2)) \Downarrow 66}$$

**Ejercicio 2.6.** Dada la siguiente expresión de **EAB** evalúa utilizando **semántica de paso grande**. Adicionalmente proporciona la representación en sintaxis abstracta.

Sintaxis Concreta:

`if ( False then (3 * 7) + 1 else (2 * 7) + 1 )`

Sintaxis Abstracta:

`if ( False, sum(prod(3, 7), 1), sum(prod(2, 7), 1) ) \Downarrow 15`

### Evaluación Paso Grande:

$$\frac{\frac{\frac{False \Downarrow False}{\text{bsbool}} \quad \frac{\frac{2 \Downarrow 2}{\text{bsnum}} \quad \frac{7 \Downarrow 7}{\text{bsnum}}}{\text{prod}(2, 7) \Downarrow 14} \quad \frac{1 \Downarrow 1}{\text{bsnum}}}{\text{bsprod}} \quad \frac{\text{sum}(\text{prod}(2, 7), 1) \Downarrow 15}{\text{bssum}}}{\text{bsiff}}$$

$$\frac{\text{if} ( False, \text{sum}(\text{prod}(3, 7), 1), \text{sum}(\text{prod}(2, 7), 1) ) \Downarrow 15}{\text{bsiff}}$$

**Nota** De este ejercicio podemos observar que dependiendo del valor que se obtenga en  $e_0$  de nuestras expresiones  $\text{if}(e_0, e_1, e_2)$  las reglas **bsiff** y **bsift** nos permiten omitir la evaluación de la expresión  $e_1$  ó  $e_2$  respectivamente, en este caso se omite la evaluación de la expresión  $e_1 = (3 * 7) + 1$ .

## 3 La función eval

Concluimos este capítulo enunciando la relación entre **semántica de paso pequeño** y **semántica de paso grande** con la función de evaluación definida para **EAB** como sigue:

**Definición 3.1.** Se define la función **eval** en términos de la semántica dinámica del lenguaje como sigue:

$$eval(e) = e_f \text{ si y sólo si } e \rightarrow^* e_f \text{ y } e_f \not\rightarrow$$

**Nota:** La propiedad de **bloqueo de valor** para **EAB** se deriva de las reglas **bsnum** y **vsbool** dónde se cumple que Si  $v$  valor entonces  $v \not\rightarrow$ , es decir  $v$  está bloqueado.

La función **eval** será de utilidad para el resto de las secciones que visitaremos a lo largo del curso.

## 4 Ejercicios para el lector

**Ejercicio 4.1.** Considera la siguiente sintaxis concreta para un lenguaje proposicional simple (*Prop*) donde solo se utiliza el conector AND ( $\wedge$ ) y el operador NOT ( $\neg$ ) definida como:

$$\frac{x \text{ Prop} \quad y \text{ Prop}}{x \wedge y \text{ Prop}} \quad \frac{x \text{ Prop}}{\neg x \text{ Prop}} \quad \frac{}{\top \text{ Prop}} \quad \frac{}{\perp \text{ Prop}}$$

- A) Proporciona una semántica de paso pequeño para evaluar las expresiones en *Prop*.  
 B) Proporciona una semántica de paso grande para evaluar las expresiones *Prop*.

**Ejercicio 4.2.** Ahora supón que se quieren añadir cuantificadores y variables a nuestro lenguaje *Prop* de la siguiente forma:

$$\frac{x \text{ Prop}}{\exists v, x \text{ Prop}} \quad \frac{x \text{ Prop}}{\forall v, x \text{ Prop}} \quad \frac{v \text{ variable}}{v \text{ Prop}}$$

Proporciona un conjunto de reglas que definen la semántica estática que nos permite decidir cuando una expresión de *Prop* no contiene variables libres bajo un contexto  $\Gamma$ . Denotado de la siguiente forma:

$$\Gamma \vdash e \text{ Ok}$$

**Ejercicio 4.3.** Una calculadora de **Notación Polaca Reversa** es una calculadora que no requiere de parentizado para evaluar las expresiones que son pasadas como argumento. Ésta se apoya de una **pila** para .“empujar” los operandos y los operadores así como de la **notación post-fija**, es decir el operador se escribe después de los operandos, por ejemplo:

$$7 + 1 = 7 1 +$$

$$7 - (3 + 2) = 7 3 2 + -$$

Esta calculadora empuja símbolos a la pila hasta encontrar un operador, en tal caso dos símbolos son sacados de la pila y el resultado de la operación es empujado. La sintaxis concreta de la calculadora está definida por las siguientes reglas:

$$\frac{x \in N}{x \text{ Symbol}} \quad \frac{x \in \{+, -, *, /\}}{x \text{ Symbol}} \quad \frac{}{NPR} \quad \frac{x \text{ Symbol} \quad xs \text{ NPR}}{x xs \text{ NPR}}$$

Esta gramática tiene el problema de poder formar expresiones que no pertenecen necesariamente a **NPR** como:

$$1 + 2$$

$$+ * /$$

- A) Proporciona un conjunto de reglas para definir la semántica estática que pueda analizar una expresión  $e$  y nos diga si es una expresión perteneciente a **NPR** denotando el juicio como:

$$\vdash e \text{ Ok}$$

B) Proporciona un conjunto de reglas para definir la semántica de paso pequeño que evalúen las expresiones de **NPR**.

C) Proporciona un conjunto de reglas para definir la semántica de paso grande que evalúen las expresiones de **NPR**.

**Ejercicio 4.4.** Utilizando el analizador sintáctico y las reglas de semántica de paso pequeño y grande definidas en el ejercicio anterior contesta lo siguiente:

Dada la expresión **NPR**

$$e = 7 \ 1 \ 9 \ - \ -$$

A) muestra que  $\vdash e \text{ Ok}$

B) Evalúa la expresión utilizando la semántica de paso pequeño.

C) Evalúa la expresión utilizando la semántica de paso grande.

**Ejercicio 4.5.** Dada la siguiente expresión de **EAB** definida como:

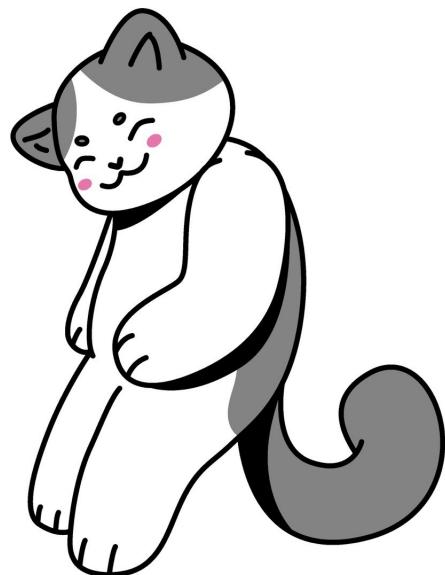
```
e = let a = let d = 2 in d + 1 end in  
    let b = (a + 1) + a in  
        (a + b) + let c = 1 in b * c end  
    end  
end
```

- Utilizando el analizador sintáctico definido para **EAB** ( $\sim$ ) decide si esta expresión es cerrada.
- Utilizando la semántica dinámica de paso pequeño definida para **EAB** muestra la evaluación de la expresión hasta obtener un valor.
- Utilizando la semántica dinámica de paso grande definida para **EAB** muestra la evaluación de la expresión hasta obtener un valor.



## Capítulo 5

### Cálculo Lambda



Es extraño concebir una vida sin teléfonos, relojes o televisores que no tengan una pequeño procesador embedido. Las computadoras son entidades relativamente nuevas pero por ajeno que pudiera parecer, hubo un tiempo en el que estas herramientas no existían, sus bases debieron ser sentadas en algún momento de nuestra historia humana.

Podemos pensar en los antiguos griegos como Euclides que escribió uno de los primeros algoritmos que nos permite encontrar el máximo común divisor de dos números naturales o algo no tan lejano, quizá más mecánico y acercado a lo que hoy entendemos como computador como la máquina analítica de Babage del siglo XIX.

Para estudiar el nacimiento de esta ciencia no debemos irnos tan atrás en la historia, hubo una época de excepcional progreso e interés en el desarrollo de las computadoras (parcial-

mente impulsado por un conflicto bélico, como muchos de los avances del siglo pasado).

Cuando pensamos en el nacimiento de las computadoras podemos inmediatamente asociarlo con las máquinas de Turing, un sistema de transición que se ha estudiado con anterioridad en materias como **Autómatas y Lenguajes Formales** siendo el más reconocido cuando se habla de modelos de computo y la definición de conceptos como **computabilidad, algoritmo ó complejidad** allá por el no tan lejano 1936.

En este curso estudiaremos un modelo equivalente (y muy distinto pero interesante) de cómputo, es aquí donde nuestro protagonista Alonzo Church<sup>1</sup> nos introduce en 1928 a su aproximación de un sistema formal basado en el concepto de "función" las primitivas de abstracción y aplicación" que fungiría como fundamento lógico para sustituir el la teoría de conjuntos de Zermelo y la teoría de tipos de Russell. Este sistema fue demostrado como inconsistente por sus dos alumnos Stephen Kleene y Barkley Russel pero no todo sería desechado, conservando la parte del manejo de funciones, particularmente rica, consistente y sorprendentemente equivalente al modelo de Alan Turing: **El Cálculo Lambda.**

## Objetivo

El interés principal de este capítulo está en revisar la composición del **Cálculo Lambda** mediante la revisión de la sintaxis, semántica operacional y las representaciones de tipos primitivos en éste modelo de cómputo así como las propiedades inherentes a la semántica del cálculo junto con los sistemas de recursión para la evaluación de las  $\lambda$ -expresiones.

## Planteamiento

En este capítulo se revisará la **sintaxis concreta** del Cálculo Lámbda<sup>2</sup> para generar  $\lambda$ -expresiones junto con La **semántica operacional** del cálculo basándonos en la operación de sustitución sintáctica ( $[:=]$  conocida como  $\beta$ -reducción).

La definición de los **booleanos** junto con los **operadores lógicos, datos estructurados** (en particular se estudiarán las tuplas, proyecciones y listas) así como **numerales** y **operadores aritméticos** mismos que se espera el lector pueda manipular, razonar y resolver en la sección de ejercicios.

Finalmente se introducirá al lector el sistema de recursión para el Cálculo Lambda con los **operadores de punto fijo**.

<sup>1</sup>Nacido en Washington en 1903 y fallecido en 1995 en Ohio, formado en la universidad de Princeton donde acogió a Alan durante su doctorado fungiendo como mentor.

<sup>2</sup>Las definiciones revisadas en este capítulo fueron extraídas de: Enríquez Mendoza J. Lenguajes de Programación Notas de Clase: Cálculo Lambda. Universidad Nacional Autónoma de México. 2022.

# 1 Sintáxis del Cálculo Lambda

El cálculo lambda es un modelo simple, su sintáxis comprende sólo tres categorías de términos:

- **Variables:** Los elementos de ésta categoría pertenecen a un conjunto finito y son las expresiones lambda más simples, se denotan igual que en álgebra como letras, generalmente las últimas del alfabeto (**w, x, y, z**). *infinito*
- **Abstracciones:** Esta categoría engloba los términos que definen a las funciones anónimas compuestas por tres elementos: el primero es la letra griega  $\lambda$ , el segundo es la variable que estará ligada en el cuerpo de la expresión y por último el cuerpo mismo de la función denotado como  $e$ , éstas se representan de la forma  $\lambda x.e$ .
- **Aplicación:** Esta categoría engloba a las expresiones que representan la aplicación de un argumento a una función. estas son representadas como  $e_1 e_2$  y ~~expresión~~ ~~sorá la sustitución sintáctica ([:-])~~ *todavía no veríamos de* *descripción* *de* *función*

**Definición 1.1** (Sintaxis concreta del Cálculo Lambda). La sintaxis concreta del Cálculo Lambda se da con la siguiente definición inductiva, sobre el juicio  $l \lambda$  que indica que  $l$  es una expresión válida en el Cálculo Lambda.

$$\frac{x \text{ var}}{x \lambda} \text{ var} \quad \frac{x \text{ var} \quad e \lambda}{\lambda x.e \lambda} \text{ abs} \quad \frac{e_1 \lambda \quad e_2 \lambda}{e_1 e_2 \lambda} \text{ app}$$

**Ejemplo 1.1.** Algunas expresiones válidas del cálculo lambda se pueden escribir de la siguiente manera:

1. x variable
2. y variable
3.  $\lambda x.x$  función anónima
4.  $\lambda x.y$  función anónima
5.  $(\lambda x.x)x$  aplicación
6.  $(\lambda x.x)y$  aplicación
7.  $\lambda x.\lambda y.xy$  currificación de dos parámetros *hay que detallar más* *en este concepto*
8.  $x(\lambda x.y)$  aplicación
9.  $(\lambda x.x)(\lambda y.y)$  aplicación de dos funciones anónimas

Vale la pena remarcar que el ejemplo 3) corresponde a la función identidad que regresa el argumento que es pasado como parámetro. El ejemplo 4) corresponde a la función constante que siempre regresa el mismo valor sin importar el parámetro que recibe. El ejemplo 7) es la ilustración de como se puede anidar dos funciones anónimas para construir una sola función de dos parámetros, a éste proceso se le conoce como **currificación**<sup>3</sup> y nos permite anidar tantas funciones como parámetros necesitemos. Dicho tema será revisado a detalle más adelante en este manual.

<sup>3</sup>En honor a Haskell Curry

## 2 $\alpha$ -equivalencia en el Cálculo Lambda

En el cálculo lambda se tiene un constructor similar al constructor `let`<sup>4</sup> que tiene una variable ligada, un alcance asociado a ella y con ésto el concepto de  **$\alpha$ -equivalencia** fue introducido para expresiones que difieren a lo más en el nombrado de sus variables ligadas. Para el cálculo lambda se tiene el mismo caso.

**Definición 2.1** ( $\alpha$ -equivalencia). En Cálculo Lambda, dos lambda términos  $e_1$  y  $e_2$  son  $\alpha$ -equivalentes si y sólo si solo se diferencian en el nombre de las variables de ligado. Ésta es denotado como:

$$e_1 \equiv_{\alpha} e_2$$

Por ejemplo, las expresiones:

$$\lambda x.x \quad \lambda z.z$$

son  $\alpha$ -equivalentes y se denota como  $\lambda x.x \equiv_{\alpha} \lambda z.z$

## 3 Semántica operacional del Cálculo Lambda

Cómo se discutió brevemente en la sección de sintáxis para el cálculo Lambda, la semántica operacional del mismo estará definida por la sustitución sintáctica, a la operación de sustituir los términos que concuerden con la variable del operador en la expresión Lambda  $e$  (redex)<sup>5</sup> := (reducto) se le conoce como  $\beta$ -reducción.

**Definición 3.1** (Semántica operacional del Cálculo Lambda). La semántica operacional del Cálculo Lambda está dada por la siguiente regla conocida como  $\beta$  reducción.

$$(\lambda x.t) s \rightarrow_{\beta} t[x := s]$$

Dónde se tienen los siguientes casos según la composición de la expresión Lambda:

- $x[x := r] = r$ .
- $y[x := r] = y$  si  $x \neq y$ .
- $(ts)[x := r] = t[x := r]s[x := r]$ .
- $(\lambda y.t)[x := r] = \lambda y.t[x := r]$  donde  $y \notin FV(r)$ .

**Nota:** la asociatividad de las aplicaciones Lambda es hacia la izquierda.

**Ejercicio 3.1.** Utiliza la definición de  $\beta$ -reducción para evaluar la siguiente expresión Lambda.

$$\begin{aligned} e &= (\lambda x.z)y \\ (\lambda x.z)y &\rightarrow_{\beta} z[x := y] \\ &= z \end{aligned}$$

abstracta

<sup>4</sup>Este constructor sirvió para presentar la sintáxis de orden superior que en el cálculo lambda tiene propiedades similares.

<sup>5</sup>este terminó es acuñado del inglés **reducible expression**.

**Ejercicio 3.2.** Utiliza la definición de  $\beta$ -reducción para evaluar la siguiente expresión Lambda.

$$\begin{aligned}
 e &= (\lambda x.x)(\lambda y.yy)z \\
 &(\lambda x.x)(\lambda y.yy)z \rightarrow_{\beta} (x[x := \lambda y.yy])(z) \\
 &= (\lambda y.yy)(z) \rightarrow_{\beta} yy[y := z] \\
 &= y[y := z]y[y := z] = zz
 \end{aligned}$$

**Ejercicio 3.3.** Utiliza la definición de  $\beta$ -reducción para evaluar la siguiente expresión Lambda.

$$\begin{aligned}
 e &= (\lambda x.\lambda y.xy)(\lambda z.z)(w) \\
 &(\lambda x.\lambda y.xy)(\lambda z.z)(w) \rightarrow_{\beta} (\lambda y.xy[x := \lambda z.z])(w) \\
 &= (\lambda y.x[x := \lambda z.z]y[y := \lambda z.z])(w) = (\lambda y.(\lambda z.z)y)(w) \\
 &(\lambda y.(\lambda z.z)y)(w) \rightarrow_{\beta} (\lambda y.z[z := y])(w) \\
 &= (\lambda y.y)(w) \rightarrow_{\beta} y[y := w] \\
 &= w
 \end{aligned}$$

## 4 Definibilidad Lambda

En el cálculo Lambda la idea principal es que las funciones son "tipos primitivos", estas pueden ser pasadas como argumentos entre sí y generar nuevos elementos válidos para el cálculo.

Si llevamos esta idea para definir entidades nos encontraremos nombrando funciones para construir cosas que damos por sentadas en los lenguajes de programación, específicamente booleanos y números.

### 4.1 Booleanos y operadores lógicos

Las constantes booleanas **True** y **False** son cosas que podemos pensar como opuestas, una es la antítesis de la otra. Tratemos de trasladar esta idea a un par de funciones que parezcan mutuamente excluyentes una de la otra.

La función que toma dos argumentos y regresa el primero es el antónimo de la función que toma dos argumentos y regresa el segundo.

$$\begin{aligned}
 &\lambda x.\lambda y.x \\
 &\lambda x.\lambda y.y
 \end{aligned}$$

Estas entidades están haciendo exactamente lo contrario que hace la otra, como las funciones son objetos primitivos en el cálculo lambda entonces definiremos este par de elementos como nuestro **True** y nuestro **False** (respectivamente).

De esta forma podemos definir funciones que a su vez, construyan la lógica booleana y nos permitan operar instrucciones de control:

**Definición 4.1** (Operadores lógicos para el cálculo Lambda). Los operadores serán funciones que reciben a las funciones **True** y **False**

1. **True:**  $\lambda x.\lambda y.x$
2. **False:**  $\lambda x.\lambda y.y$
3. **NOT:**  $\lambda z.z \text{ False True}$
4. **AND:**  $\lambda x.\lambda y.xy \text{ False}$

Para el operador **NOT** la idea es que dependiendo del argumento se regrese su contrario, si el parámetro de entrada es **True** entonces se regresará el primer argumento que entra a la función, en este caso los parámetros de entrada son: **False** y **True** regresándonos el primero: **False**. Sí al contrario el parámetro de entrada es **False** éste regresará el segundo argumento de la entrada: **False** y **True**, en este caso: **True**.

En el caso del operador **AND** la idea subyacente de este operador es que si el primer parámetro es un **False** se regrese su segundo argumento, en este caso recibe: " $y\tau$  **False**" evalúandose así a: **False**. Sí recibe como primer argumento **True** este regresa el primer argumento, en este caso " $y\tau$  **False**" evalúandose a " $y$ ".

**Definición 4.2** (Sémantica Operacional de las instrucciones lógicas para el Cálculo Lambda). Con los operadores de la definición anterior es posible verificar que la siguiente semántica operacional es válida:

1.  $\text{if } true e_1 e_2 \rightarrow_{\beta}^* e_1$
2.  $\text{if } false e_1 e_2 \rightarrow_{\beta}^* e_2$
3.  $\text{not } true \rightarrow_{\beta}^* false$
4.  $\text{not } false \rightarrow_{\beta}^* true$
5.  $\text{and } false b \rightarrow_{\beta}^* false$
6.  $\text{and } true b \rightarrow_{\beta}^* b$

**Nota:** En esta definición se hace abuso de notación al escribir **True** y **False** como constantes y no como funciones, esto será permitido para

## 5 Aritmética del Cálculo Lambda

En los lenguajes de programación el tipo primitivo más importante que nos es proporcionado (a parte de los booleanos) son los números. Éstos muchas veces están categorizados en: flotantes, enteros, enteros largos, etc.

Estas entidades también tienen una representación en el cálculo Lambda, empezando por los números más simples: los naturales.

El cálculo Lambda tiene la particularidad de modelar a los naturales de una forma muy simple y cuyo mapeo a la estructura revisada en cursos como **Álgebra Superior** y **Estructuras Discretas** es inmediata. Recordemos que en estos cursos los números no son

presentados como caracteres arábigos (0,1,2,3,4 ...) si no como la aplicación de la función sucesor **S(n)** al constructor **Zero** tantas veces como el número que se desea construir, si dicho número es el 4 entonces su representación será la función sucesor aplicada cuatro veces al **Zero**: **S(S(S(S(Zero))))**

## 5.1 Numerales de Church

Church introdujó los numerales como la abstracción de dos parámetros **s** y **z** en una función anónima, de tal forma que si se desea representar al n-ésimo número éste sea formado por la aplicación de **s** a **z** n veces:

- $\bar{0} =_{def} \lambda s. \lambda z. z$
- $\bar{1} =_{def} \lambda s. \lambda z. sz$
- $\bar{2} =_{def} \lambda s. \lambda z. s(sz)$
- $\bar{n} =_{def} \lambda s. \lambda z. \underbrace{s(\dots(s z) \dots)}_{n \text{ veces}}$

## 5.2 Funciones aritméticas

### Función Sucesor

Al igual que los números naturales, los numerales de Chruch tienen funciones aritméticas asociadas que nos permitirán computar valores, la función aritmética más simple para los naturales es la función sucesor. Para el cálculo lambda ésta está definida como:

**Definición 5.1** (Función Sucesor definida para los numerales de Church).

$$\mathbf{suc} \quad \lambda n \lambda Z \lambda s. s(n z s)$$

la aplicación de "sz" <sup>a</sup> un numeral de Church como la que aparece en la parte interna de la función **suc** (**n s z**) reduce el numeral a una representación más parecida a la de los números naturales:

- $(\lambda z. \lambda s. z) sz \rightarrow_{\beta}^{*} z$
- $(\lambda z. \lambda s. sz) sz \rightarrow_{\beta}^{*} sz$
- $(\lambda z. \lambda s. s(sz)) sz \rightarrow_{\beta}^{*} ssz$
- ...

Junto con la aplicación de "s" que está fuera de ésta sección (**s (n s z)**) se agrega un símbolo sucesor más al numeral obteniendo así el siguiente elemento.

A continuación enlistamos los ejercicios para exemplificar la semántica operacional del operador **suc** (El abuso de notación escribiendo los numerales de Church con una línea

encima del numero que corresponde es permitida para ahorrar espacio, lo mismo es válido para las constantes booleanas **True** y **False**).

**Ejercicio 5.1.** Utiliza la definición de la función sucesor evaluar la siguiente  $\lambda$ -expresión

**suc**  $\bar{2}$

$$\begin{aligned} (\lambda n \lambda z \lambda s. s(n z s)) \lambda z' \lambda s'. s'(s' z') &\rightarrow_{\beta} \lambda z \lambda s. s((\lambda z' \lambda s'. s'(s' z')) s z) \\ &\rightarrow_{\beta} \lambda z \lambda s. s(\lambda z'. s(sz) z) \rightarrow_{\beta} \lambda s \lambda s. s(s(sz)) \\ &= \bar{3} \end{aligned}$$

**Ejercicio 5.2.** Utiliza la definición de la función sucesor evaluar la siguiente  $\lambda$ -expresión

**suc**  $\bar{5}$

$$\begin{aligned} (\lambda n \lambda z \lambda s. s(n z s)) \lambda z' \lambda s'. s'(s'(s'(s'(s' z')))) &\rightarrow_{\beta} \lambda z \lambda s. s((\lambda z' \lambda s'. s'(s'(s'(s' z')))) s z) \\ &\rightarrow_{\beta} \lambda z \lambda s. s(\lambda z'. s(s(s(s(z')))) z) \rightarrow_{\beta} \lambda s \lambda z. s(s(s(s(s(z)))))) \\ &= \bar{6} \end{aligned}$$

## Is Zero

Este operador sirve para comprobar si un numeral es cero, regresa el booleano **True** en caso de que el numeral corresponda y **False** cuando no.

**Definición 5.2** (Función IsZero para numerales de Church.).

**IsZero:**  $\lambda m. m \text{ True } (\lambda x. \text{False})$

**Ejercicio 5.3.** Utiliza las definición de la función **IsZero** para evaluar la siguiente  $\lambda$ -expresión

**IsZero:**  $\bar{2}$

$$\begin{aligned} (\lambda m. m \text{ True } (\lambda x. \text{False})) \lambda z \lambda s. s(s z) &\rightarrow_{\beta} (\lambda z \lambda s. s(s z)) \text{ True } (\lambda x. \text{False}) \\ (\lambda s. s(s \text{ True})) (\lambda x. \text{False}) &\rightarrow_{\beta} (\lambda x. \text{false}) ((\lambda x. \text{false}) \text{ true})) \\ &\rightarrow_{\beta} (\lambda x. \text{false}) \text{ false } \rightarrow_{\beta} \text{ False} \end{aligned}$$

**Ejercicio 5.4.** Utiliza las definición de la función **IsZero** para evaluar la siguiente  $\lambda$ -expresión

**IsZero:**  $\bar{0}$

$$\begin{aligned} (\lambda m. m \text{ True } (\lambda x. \text{False})) \lambda z \lambda s. z &\rightarrow_{\beta} (\lambda z \lambda s. z) \text{ True } (\lambda x. \text{False}) \\ &\rightarrow_{\beta} (\lambda s. \text{True}) \lambda x. \text{False } \rightarrow_{\beta} \text{ True} \end{aligned}$$

## Suma

La suma para los números naturales está definida por inducción sobre el segundo argumento cuyos casos son:

1.  $m + 0 = m$
2.  $m + S n = S(m + n)$

Para el cálculo lambda la definición de esta función será:

**Definición 5.3** (Definición de la suma para los numerales de Church).

$$\text{sum: } \lambda m \lambda n \lambda z \lambda s. n(m z s) s$$

la definición de esta función puede ser entendida en términos de la función suma para los naturales. El argumento **m** no es afectado por la función en ningún momento, ésto se regleja en la sección interna de nuestra lambda, a saber (**m z s**). Ésto no altera **m**, lo simplifica pero mantiene la cantidad de aplicaciones de **s**, esta simplificación de **m** es el primero argumento que va a recibir el numeral **n** tomando el lugar de **z**.

El segundo argumento del numeral **n** será la **s** que simplemente reemplazara todas las apariciones de **s** por **s** misma.

De esta forma lo que al final sucede es que se concatenarán las "s's" de ambos numerales generando así el numeral correspondiente a la suma de las apariciones del carácter "s." en ambos **n** y **m**.

**Ejercicio 5.5.** Utiliza la definición de la función **sum** evaluar la siguiente  $\lambda$ -expresión

$$\text{sum } \bar{2} \bar{3}$$

$$\begin{aligned}
&= (\lambda m \lambda n \lambda z \lambda s. n(m z s) s) \lambda z' \lambda s'. s'(s' z') \lambda z'' \lambda s''. s''(s''(s'' z'')) \rightarrow_{\beta} \\
&\quad (\lambda n \lambda z \lambda s. n((\lambda z' \lambda s'. s'(s' z')) z s) s) \lambda z'' \lambda s''. s''(s''(s'' z'')) \rightarrow_{\beta} \\
&\quad \lambda z \lambda s. ((\lambda z'' \lambda s''. s''(s''(s'' z''))) ((\lambda z' \lambda s'. s'(s' z')) z s) s) \rightarrow_{\beta} \\
&\quad \lambda z \lambda s. (\lambda z'' \lambda s''. s''(s''(s'' z''))) ((\lambda s'. s'(s' z)) s) s) \rightarrow_{\beta} \\
&\quad \lambda z \lambda s. (\lambda z'' \lambda s''. s''(s''(s'' z''))) (s(sz)) s) \rightarrow_{\beta} \\
&\quad \lambda z \lambda s. (\lambda s''. s''(s''(s(sz)))) s) \rightarrow_{\beta} \\
&\quad \lambda z \lambda s. s(s(s(sz))) = \bar{5}
\end{aligned}$$

**Ejercicio 5.6.** Utiliza la definición de la función **sum** evaluar la siguiente  $\lambda$ -expresión

$$\text{sum } \bar{4} \bar{5}$$

$$\begin{aligned}
&= (\lambda m \lambda n \lambda z \lambda s. n(m z s) s) \lambda z' \lambda s'. s'(s'(s' z')) \lambda z'' \lambda s''. s''(s''(s''(s'' z''))) \rightarrow_{\beta} \\
&\quad (\lambda n \lambda z \lambda s. n((\lambda z' \lambda s'. s'(s'(s' z')) z s) s) \lambda z'' \lambda s''. s''(s''(s''(s'' z''))) \rightarrow_{\beta}
\end{aligned}$$

$$\begin{aligned}
& \lambda z \lambda s (\lambda z'' \lambda s''. s''(s''(s''(s''z'')))) (\lambda z' \lambda s'. s'(s'(s'z'))) z s) s) \rightarrow_{\beta} \\
& \lambda z \lambda s ((\lambda z'' \lambda s''. s''(s''(s''(s''z'')))) (\lambda s'. s'(s'(s'z))) s) s) \rightarrow_{\beta} \\
& \lambda z \lambda s (\lambda z'' \lambda s''. s''(s''(s''(s''z'')))) s(s(s(sz))) s) \rightarrow_{\beta} \\
& \lambda z \lambda s (\lambda s''. s''(s''(s''(s''(s(s(s(sz)))))))) s) \rightarrow_{\beta} \\
& \lambda s. \lambda z. s(s(s(s(s(s(sz))))))) = \bar{9}
\end{aligned}$$

## Producto

Para el producto de **m** y **n** la idea es anidar la operación: **n + n**, **m** veces reemplazando las apariciones de la variable **s** en el numeral **m** por **sum(n ...)**, esta operación está definida de la siguiente manera:

**Definición 5.4** (Definición del producto para los numerales de Church.).

$$\text{prod: } \lambda m. \lambda n. m \bar{0} (\text{sum } n)$$

**Ejercicio 5.7.** Utiliza la definición de la función **prod** evaluar la siguiente  $\lambda$ -expresión

$$\text{prod } \bar{2} \bar{3}$$

$$\begin{aligned}
& = (\lambda m \lambda n. m \bar{0} (\text{sum } n)) \lambda z \lambda s. s(sz) \bar{3} \rightarrow_{\beta} (\lambda n. (\lambda z \lambda s. s(sz)) \bar{0} (\text{sum } n)) \bar{3} \\
& \rightarrow_{\beta} (\lambda n. (\lambda s. s(s\bar{0})) (\text{sum } n)) \bar{3} \rightarrow_{\beta} (\lambda n. (\text{sum } n (\text{sum } n \bar{0})) \bar{3} \\
& \rightarrow_{\beta} (\text{sum } \bar{3} (\text{sum } \bar{3} \bar{0})) =
\end{aligned}$$

$$\lambda m \lambda n \lambda s \lambda z. n(m z s) s \lambda z' \lambda s'. s'(s'(s'z')) (\lambda m' \lambda n' \lambda s'' \lambda z'' . n'(m' z'' s'') s \lambda z''' \lambda s''' . s'''(s'''(s'''z''')) \lambda z'''' \lambda s'''' . z''''')$$

resolveremos por partes la suma dado que la notación crece muy rápido y es poco legible

$$\begin{aligned}
\text{sum } \bar{3} \bar{0} & = (\lambda m \lambda n \lambda z \lambda s. n(m z s) s) \lambda z' \lambda s'. z' \bar{3} \\
& \rightarrow_{\beta} (\lambda n \lambda z \lambda s. n((\lambda z' \lambda s'. z') z s) s) \bar{3} \\
& \rightarrow_{\beta} (\lambda n \lambda z \lambda s. n(\lambda s'. z s) s) \bar{3} \rightarrow_{\beta} (\lambda n \lambda z \lambda s. n z s) \bar{3} \\
& \rightarrow_{\beta} \lambda z \lambda s. (\bar{3} z s) = \lambda z \lambda s. ((\lambda z' \lambda s'. s'(s'z'))) z s \\
& \rightarrow_{\beta} \lambda z \lambda s. (\lambda s'. s'(s'(z))) s) \rightarrow_{\beta} \lambda z \lambda s. s(sz) = \bar{3} \\
\text{sum } \bar{3} \bar{3} & = (\lambda m \lambda n \lambda z \lambda s. n(m z s) s) \lambda z' \lambda s'. s'(s'(s'. z')) \bar{3} \\
& \rightarrow_{\beta} (\lambda n \lambda z \lambda s. n((\lambda z' \lambda s'. s'(s'. z')) z s) s) \bar{3} \\
& \rightarrow_{\beta} (\lambda n \lambda z \lambda s. n((\lambda s'. s'(s'. z')) s) s) \bar{3} \\
& \rightarrow_{\beta} (\lambda n \lambda z \lambda s. n(s(s(s.z))) s) \bar{3} \\
& \rightarrow_{\beta} \lambda z \lambda s. ((\bar{3}) (s(s(s.z))) s) = \lambda m \lambda n \lambda z \lambda s. ((\lambda z' \lambda s'. s'(s'z')) (s(s(s.z))) s) \\
& \rightarrow_{\beta} \lambda z \lambda s. ((\lambda s'. s'(s'(s(s(s.z)))))) s) \\
& \rightarrow_{\beta} \lambda z \lambda s. \lambda s. s(s(s(s(s.z)))) = \bar{6}
\end{aligned}$$

# 6 Datos estructurados en el Cálculo Lambda

En el **Cálculo Lambda** es posible definir estructuras para almacenar información junto con las funciones para recuperar los elementos guardados en éllas. Revisaremos dos: La estructura más simple serán las **tuplas** junto con sus proyecciones **first** y **second** que ayudan a sentar la base para definir una estructura más compleja, las listas. Con su respectiva función para obtener la **cabeza** y la **cola**. Juntas proveen un mecanismo de almacenamiento en este sistema.

## 6.1 Tuplas

La tupla es la estructura que representa a un par compuesto por elemento izquierdo y derecho o primero y segundo. En el cálculo Lambda se define como la función que tiene dos argumentos uno para cada elemento. Y una función **b** (el argumento **f** representa al elemento primero o *first* en inglés y el argumento **s** representa el segundo o *second*).

**Definición 6.1** (Constructor de una tupla junto con la proyección de ambos elementos).  
**Constructor**

$$\text{pair} = \lambda f \lambda s \lambda b. b f s$$

**Proyección del primer elemento**

$$\text{fst} = \lambda p. p \text{ True}$$

**Proyección del segundo elemento**

$$\text{snd} = \lambda p. p \text{ False}$$

Donde las proyecciones se apoyan de la propiedad de los booleanos de regresar el primer argumento para el caso de **True** y el segundo en el caso de **False**<sup>6</sup>.  
En general se tienen las siguientes reducciones cuando se aplican estos operadores a cualquier tupla:

$$\text{fst } (\text{pair } a \ b) \rightarrow_{\beta}^{*} a$$

$$\text{snd } (\text{pair } a \ b) \rightarrow_{\beta}^{*} b$$

## 6.2 Listas

Las listas en el Cálculo Lambda se apoyan de los constructores de la tupla para ir añadiendo elementos en la cabeza al cuerpo de una lista anidando las tuplas un elemento a la vez. El único inconveniente es que no hay un constructor para la lista vacía, éste será representado por una  $\lambda$ -expresión que sea  $\alpha$ -equivalente al booleano **False**.

<sup>6</sup>Véase la sección de este capítulo: **Booleanos**

La definición de las listas resulta ser una aplicación de los constructores para tuplas, sus funciones y la constante booleana **False** como se ve a continuación:

**Definición 6.2** (Constructores para listas en Cálculo Lambda junto con sus funciones).

**Definición de la Lista vacía**

$$\text{nil} = \text{False}$$

**Constructor de una lista tomando una cabeza y una cola**

$$\text{cons} = \text{pair}$$

**función para obtener la cabeza de la lista**

$$\text{head} = \text{fst}$$

**función para obtener la cola de la lista**

$$\text{tail} = \text{snd}$$

## 7 Propiedades semánticas del Cálculo Lambda

En el capítulo 5. **Sémantica** se mencionaron brevemente las propiedades de la evaluación para expresiones de **EAB**. En esta sección discutiremos cómo el Cálculo Lambda interactúa con las mismas propiedades con la ayuda de ejercicios que nos permitan entender cada propiedad aplicada en este modelo.

### 7.1 No terminación

Hemos tratado brevemente las propiedades de la evaluación para expresiones bien formadas de **EAB** en donde éstas se evaluarán a un valor (número o booleano) en algún punto de su ejecución. No obstante esta propiedad no se cumple para el Cálculo Lambda como se puede ver a continuación:

**Ejercicio 7.1.** Demuestra o da un contraejemplo de por qué la propiedad de terminación para el Cálculo Lambda es válida.

Consideremos el siguiente par de  $\lambda$ -expresiones:

$$\omega = \lambda x.xx$$

$$\Omega = \omega\omega$$

$$\Omega = \omega\omega = (\lambda x.xx)\omega\omega \rightarrow_{\beta} \omega\omega$$

En general esta expresión cumple que en cada  $\beta$ -reducción se tiene la misma expresión  $\Omega$ . Por lo tanto una expresión bien formada del Cálculo Lambda no necesariamente tiene una forma normal.

## 7.2 No determinismo

En los lenguajes de programación una propiedad deseable es la propiedad determinista de los programas, es decir, que la evaluación que se haga para una expresión sea siempre la misma. El Cálculo Lambda carece de esta propiedad. Dependiendo del segmento *redex* que escogamos para aplicar la  $\beta$ -reducción el reducto puede diferir, tomemos como ejemplo la siguiente  $\lambda$ -expresión:

$$(\lambda z.(\lambda y.z)a)b$$

Si tomamos como redex la expresión completa, se obtiene la siguiente evaluación:

$$(\lambda z.(\lambda y.z)a)b \xrightarrow{\beta} (\lambda y.b)a$$

Si por el contrario tomamos como redex la lambda interna se obtiene:

$$(\lambda z.(\lambda y.z)a)b \xrightarrow{\beta} (\lambda z.z)b$$

En ambas evaluaciones el resultado al que se llega al terminar de evaluar la expresión es la variable **b** sin importar cuál redex se haya escogido para la evaluación.

Esta característica del Cálculo Lambda no supone una desventaja por el principio de **confluencia**.

## 7.3 Confluencia

El principio de confluencia nos asegura que dadas dos evaluaciones distintas para la misma  $\lambda$ -expresión, éstas convergen en un término común en algún punto de la evaluación:

**Teorema 7.1** (Propiedad de Church-Rosser). Si  $e \xrightarrow{\beta}^* e_1$  y  $e \xrightarrow{\beta}^* e_2$  entonces existe un término  $t$  tal que  $e_1 \xrightarrow{\beta}^* t$  y  $e_2 \xrightarrow{\beta}^* t$ .

**Corolario 7.2** (Unicidad de formas normales). Para cualquier expresión  $e$  si  $e \xrightarrow{\beta}^* e_f$  y  $e \xrightarrow{\beta}^* e'_f$  tal que tanto  $e_f$  como  $e'_f$  están bloqueadas, entonces  $e_f = e'_f$  salvo  $\alpha$ -equivalencias. Es decir, la forma normal de una expresión es única.

## 8 Combinadores de punto fijo

El Cálculo Lambda por si mismo no posee un mecanismo iterativo que nos permita formar alguna secuencia de control como los ciclos *for* o *while* como en la mayoría de los lenguajes de programación modernos.

Su naturaleza funcional nos hace preguntarnos si podemos emplear un mecanismo más afín como la **recursión** para definir funciones que iteren sobre algún parámetro.

El mecanismo de recursión para el Cálculo Lambda no viene dado por la definición que hasta el momento hemos revisado, se precisa de la introducción de los **Combinadores de Punto Fijo** que capturan la esencia del principio de recursión general en computación, ésto es:

$$\text{rec } F = F(\text{rec } F)$$

De esta forma, si definimos una  $\lambda$ -expresión rec que nos permita “autoaplicarse” la función que toma como parámetro podemos definir cualquier función recursiva aplicando n veces la función.

$$\text{rec } F = F(\text{rec } F) = F(F(\text{rec } F)) = \dots F(F(F(\dots)))$$

**Definición 8.1** (Combinador de punto fijo). Un lambda término cerrado  $F$  es un combinador de punto fijo si y sólo si cumple alguna de las siguientes condiciones:

1.  $Fg \rightarrow_{\beta}^{*} g(Fg)$
2.  $Fg \equiv_{\beta} g(Fg)$ , es decir, existe un término  $t$  tal que  $Fg \rightarrow_{\beta}^{*} t$  y  $g(Fg) \rightarrow_{\beta}^{*} t$

Existen diferentes  $\lambda$ -expresiones que cumplen con la definición provista anteriormente, uno de los combinadores mas sencillos y populares es el Combinador  $\mathbb{Y}$  que se define de la siguiente forma:

**Definición 8.2.** (Combinador  $\mathbb{Y}$ ) también conocido como **Curry-Rosser**:

$$\mathbb{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

**Ejercicio 8.1.** Demuestra que el combinador  $\mathbb{Y}$  es un combinador de punto fijo.

$$\begin{aligned} & \mathbb{Y}g \\ & (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))g \\ & \rightarrow_{\beta} (\lambda x.g(xx))(\lambda x.g(xx)) \\ & \rightarrow_{\beta} g((\lambda x.g(xx))(\lambda x.g(xx))) \end{aligned}$$

Tomando la segunda parte de la igualdad y desarrollando obtenemos

$$\begin{aligned} & g(\mathbb{Y}g) \\ & g((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))g) \\ & \rightarrow_{\beta} g((\lambda x.g(xx))(\lambda x.g(xx))) \end{aligned}$$

de esta forma podemos concluir que  $\mathbb{Y}g \equiv_{\beta} g(\mathbb{Y}g)$ , entonces  $\mathbb{Y}$  es un combinador de punto fijo.

## 9 Ejercicios para el lector

### - Ejercicio para identificar λ-term

**Ejercicio 9.1.** Continuando con la definición 4.1 de los booleanos y sus operadores en el Cálculo Lambda implementa los siguientes operadores:



¿Qué deben de hacer?

**Ejercicio 9.2.** Verifica las propiedades de la semántica operacional para cada operador booleano según la definición 4.2:

$$\begin{aligned} \text{if } true\ e_1\ e_2 &\rightarrow_{\beta}^{*} e_1 \\ \text{if } false\ e_1\ e_2 &\rightarrow_{\beta}^{*} e_2 \\ \text{not } true &\rightarrow_{\beta}^{*} false \\ \text{not } false &\rightarrow_{\beta}^{*} true \\ \text{and } false\ b &\rightarrow_{\beta}^{*} false \\ \text{and } true\ b &\rightarrow_{\beta}^{*} b \\ \text{or } true\ b &\rightarrow_{\beta}^{*} true \\ \text{or } false\ b &\rightarrow_{\beta}^{*} b \end{aligned}$$

**Ejercicio 9.3.** Resuelve las siguientes operaciones de los númerales de Church (el abuso de notación para simplificar λ-expresiones está permitido con el fin de tener una representación más compacta cuando sea posible)

tipo de letra

**Ejercicio 9.4.** Siguiendo la definición 6.2 para Listas en el Cálculo Lambda define la función *IsNil* que se comporta de la siguiente forma

$$\begin{aligned} \underline{\text{IsNil } [] = \text{True}} \\ \underline{\text{IsNil } (x : xs) = \text{False}} \end{aligned}$$

**Ejercicio 9.5.** Encuentra la forma normal de las siguientes λ-expresiones, si no es posible explica por qué.

**Ejercicio 9.6.** Para cada una de las siguientes  $\lambda$ -expresiones demuestra que son combinadores de punto fijo.

- **Turing** =  $UU$  en donde  $U = \lambda f. \lambda x. x(ffx)$
- **Estricto Z** =  $\lambda f. (\lambda x. f(\lambda v. xxv))(\lambda x. f(\lambda v. xxv))$

**Ejercicio 9.7.** Utilizando cualquiera de los combinadores de punto fijo implemeta la función recursiva **Factorial** para los numerales de Church en el Cálculo Lambda.

$$\text{Factorial } 1 = 1$$

$$\text{Factorial } n = n * \text{factorial } n - 1$$

Adicionalmente muestra la ejecución para  $n = \bar{3}$

**Ejercicio 9.8.** Utilizando algún combinador de punto fijo implemeta la función recursiva **Fibonacci** para los numerales de Church en el cálculo Lambda.

$$\text{Fibonacci } 0 = 0$$

$$\text{Fibonacci } 1 = 1$$

$$\text{Fibonacci } n = \text{Fibonacci } n - 1 + \text{Fibonacci } n - 2$$

Adicionalmente muestra la ejecución para  $n = \bar{3}$

**Ejercicio 9.9.** Utilizando el combinador Y, define la función **concat** para la representación de listas en el Cálculo Lambda.

$$\text{concat } [] (y : ys) = (y : ys)$$

$$\text{concat } (x : xs) (y : ys) = x : \text{concat}(xs (y : ys))$$

**Ejercicio 9.10.** Utilizando cualquier combinador de punto fijo resuelve lo siguiente:

1. Define la función **reversa** para las listas del Cálculo Lambda.

$$\begin{aligned} \text{rev } [] &= [] \\ \text{rev } (x : xs) &= \text{rev } xs ++ [x] \end{aligned}$$

2. Aplica la función para obtener la reversa de la lista: **pair 1 (pair 2 (pair 3 False))**

**Ejercicio 9.11.** Utilizando el combinador Y contesta lo siguiente:

1. Da una definición de la función de exponentiación para los numerales de Church.
2. Resuelve la función para **exp  $\bar{4} \bar{2}$**

# Representación de Árboles Binarios

# Capítulo 6

## MinHs



Solo hemos visto EAB

Con la teoría que hemos revisado hasta el momento en nuestro lenguaje **EAB** que incluye: *Booleanos* y *Numeros Natures*, los operadores: +, \*, <, las operaciones lógicas: **If ... then ... else**, el **Cálculo Lambda** y que junto con la definición de la semántica estática y dinámica constituyen un lenguaje funcional, nos es natural preguntarnos en como sería la implementación de un lenguaje de características similares que se ejecutase en una computadora.

Un lenguaje de programación funcional que existe (y del cual este manual bebe gran inspiración) es **Haskell** que tiene características deseables para nuestros lenguajes teóricos **EAB**.

Hasta el momento se ha trabajado con tipos de datos primitivos, pero no hemos hablado del **tipado** de los datos persé y mucho menos del tipado de las expresiones, ésto constituye un problema al poder escribir expresiones de **EAB** correctas pero cuya evaluación no termine en un valor puesto que la ejecución se detendría en algún momento al no hallar una regla de la semántica dinámica que nos permita continuar.

Con ésto en mente el objetivo de este capítulo será definir un lenguaje similar que nos permite tener todos los tipos de datos de **EAB**, el sistema de tipos de **Haskell** y las características más importante del mismo, concretamente: evaluación perezosa, pureza funcional y tipado explícito y estático.

Éste lenguaje lo llamaremos **MinHaskell** al ser menos robusto que su padre pero servirá para ilustrar los conceptos que hasta ahora hemos trabajado en este manual.

## Objetivo

El objetivo de este capítulo será proveer la definición del lenguaje funcional **MinHaskell** que preserve las características principales del lenguaje de programación **Haskell** a manera de ilustrar una implementación concreta de los conceptos que se han revisado hasta este momento: Sintáxis, Semántica, Cálculo Lambda y recursión, haciendo especial enfasis en el sistema de tipos para este lenguaje y sus propiedades.

## Planteamiento

El desarrollo del capítulo se plantea de forma similar a como lo hemos hecho con el lenguaje **EAB** dividiendo su definición en **sintáxis concreta**, **sintáxis abstracta**, **semántica dinámica** y es aquí en donde se introducirá una nueva capa para estudiar las propiedades de las expresiones tipadas: **el sistema de tipos**. Finalmente concluiremos mencionando brevemente las propiedades que este pequeño lenguaje posee<sup>1</sup>.

# 1 Sintáxis de MinHaskell

## 1.1 Sintáxis concreta

Para construir el sistema de tipos de **MinHaskell**, se necesita introducir una nueva expresión de tipo al cual las variables pertenecen que será representado por la letra **T**. El tipo de las  $\lambda$ -expresiones será representado por la expresión  $T_1 \Rightarrow T_2$  en la **Sintáxis**

---

<sup>1</sup>Las definiciones que se revisan en este capítulo concernientes a la **Sintáxis**, **Semántica** y propiedades de **MinHaskell** fueron extraídas del texto original: Enríquez Mendoza J., Lenguajes de Programación Nota de clase: Min-Haskell, Universidad Nacional Autónoma de México. 2022. Agregando la categoría de valores, las variables de función.

**Concreta** donde  $T_1$  representa el tipo de la variable ligada en el cuerpo de la expresión y  $T_2$  es el tipo que regresa la evaluación de la expresión completa.

Finalmente se introduce la expresión para funciones recursivas representadas por la expresión  $\text{letrec } var = e_1 \text{ in } e_2 \text{ end}$

**Definición 1.1** (Sintaxis Concreta de *MinHaskell*).

<b>Expresiones</b>	$e ::= var \mid n \mid b \mid (e) \mid e_1 \otimes e_2 \mid e_1 e_2$ $\mid if e_1 \text{ then } e_2 \text{ else } e_3$ $\mid let x = e_1 \text{ in } e_2 \text{ end}$ $\mid lam x :: T \Rightarrow e$ $\mid recfun f :: (T_1 \rightarrow T_2) x \Rightarrow e$
<b>Tipos</b>	$T ::= \text{Bool} \mid \text{Nat} \mid T_1 \Rightarrow T_2$
<b>Variables</b>	$var ::= x \mid y \mid \dots$
<b>Números</b>	$n ::= 0 \mid 1 \mid \dots$
<b>Booleanos</b>	$b ::= \text{True} \mid \text{False}$
<b>Operadores Infijos</b>	$\otimes ::= + \mid * \mid - \mid = \mid < \mid > \mid \geq \mid \leq$

**Ejercicio 1.1.** Escribe la definición de los siguientes programas utilizando la sintáxis anteriormente definida para *MinHaskell*

1. Escribe la función sucesor

$$lam\ x\ ::\ \text{Nat} \rightarrow x + 1$$

2. Escribe la función IsZero

$$lam\ x\ ::\ \text{Nat} \rightarrow if\ (x\ ==\ 0)\ then\ \text{True}\ else\ \text{False}$$

3. Define la función factorial

$$recfun\ fact\ ::\ (\text{Nat} \rightarrow \text{Nat})\ x \rightarrow if\ IsZero(x)\ then\ 1\ else\ n\ * fact\ (n - 1)$$

4. Define una expresión para aplicar la función sucesor utilizando el operador **let**

$$let\ f\ ::\ (\text{Nat} \rightarrow \text{Nat})\ in\ lam\ x\ ::\ \text{Nat} \rightarrow f\ x\ end$$

5. Define una función de dos parámetros para multiplicar números.

$$lam\ x\ ::\ \text{Nat} \rightarrow lam\ y\ ::\ \text{Nat} \rightarrow x * y$$

Del ejercicio anterior podemos comprobar que la sintáxis es muy similar a la de **Haskell** acercándose más a la representación que tendrían las definiciones equivalentes en el compilador **GHCi**.

## 1.2 Sintaxis abstracta

Una vez definidas las reglas para generar programas en *MinHaskell* empleando la **sintaxis concreta** del lenguaje, podemos definir su representación intermedia empleando los **árboles sintácticos** mediante la definición de las reglas generarlos a continuación enlistadas:

**Definición 1.2** (Sintaxis abstracta de *MinHaskell*). La sintaxis abstracta se define con las siguientes reglas:

### Valores y variables

$$\frac{n \in \mathbb{N}}{\text{num}[n] \text{ asa}} \quad \frac{}{\text{bool}[\text{True}] \text{ asa}} \quad \frac{}{\text{bool}[\text{False}] \text{ asa}}$$

$$\frac{x \in \text{Variables}}{x \text{ asa}}$$

### Operadores

$$\frac{t_1 \text{ asa} \quad \dots \quad t_n \text{ asa}}{o(t_1, \dots, t_n) \text{ asa}}$$

### Condicional

$$\frac{t_1 \text{ asa} \quad t_2 \text{ asa} \quad t_3 \text{ asa}}{\text{if}(t_1, t_2, t_3) \text{ asa}}$$

### Asignaciones locales

$$\frac{t_1 \text{ asa} \quad t_2 \text{ asa}}{\text{let}(t_1, x.t_2) \text{ asa}}$$

### Definición de funciones

$$\frac{t \text{ asa}}{\text{lam}(\mathbf{T}, x.t) \text{ asa}} \quad \frac{t \text{ asa}}{\text{recfun}(T, f.x.t) \text{ asa}}$$

### Aplicación de función

$$\frac{t_1 \text{ asa} \quad t_2 \text{ asa}}{\text{app}(t_1, t_2) \text{ asa}}$$

### Operador de punto fijo

$$\frac{t \text{ asa}}{\text{fix}(\mathbf{T}, f.t) \text{ asa}}$$

**Observaciones:** El operador de punto fijo **fix** es una implementación interna para evaluar expresiones recursivas. Como no está asociado a ninguna expresión de la sintaxis concreta es imposible que un usuario lo pueda instanciar directamente.

El operador **recfun** tiene dos variables ligadas en el cuerpo de la definición: el nombre de la función y la variable que recibe como argumento.

## 2 Semántica de MinHaskell

### 2.1 Sistemas de tipos

- Antes de comenzar a hablar de un sistema de tipos debemos contestar a la pregunta:
- ¿Qué es un tipo?**. Un tipo es la descripción abstracta de una colección de valores que nos permite agruparlos y emplearlos de manera similar aún sin saber el valor específico que éste pueda ser.

En los lenguajes de programación los sistemas de tipos brindan información adicional sobre la evaluación de las expresiones, qué valores debemos esperar recibir y regresar al concluir la ejecución de nuestro programa y brindan información adicional para definir restricciones que nos permitan tener seguridad y congruencia en los datos empleando una colección de reglas de tipado.

Este sistema estará embedido en la siguiente capa de MinHaskell de forma similar como fue trabajado con anterioridad con el lenguaje **EAB**, dicho nivel es la **semántica estática** que define un conjunto de juicios para brindar la seguridad de evaluación a las expresiones.

### 2.2 Semántica estática

**Haskell** es un lenguaje de programación categorizado como **fuertemente tipado**, ésto quiere decir que la evaluación de las expresiones solo es posible cuando el programa es congruente con las reglas definidas por su sistema de tipos, descartando la evaluación de todas aquellas expresiones que estén bien formadas pero que no respeten las restricciones del sistema.

no evalua

La **semántica estática** nos ayuda a definir criterios (juicios) para evaluar los programas de MinHaskell con la información que se pueda inferir acerca de los parámetros que una expresión toma como argumento o el tipo que ésta regresa al concluir su evaluación.

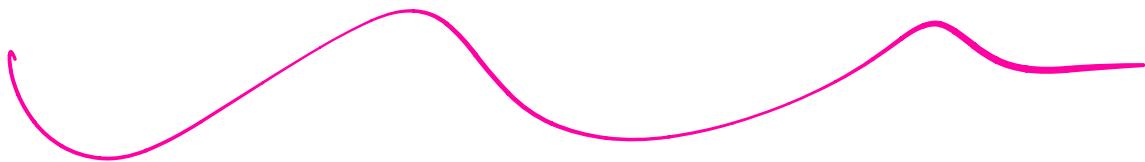
**Definición 2.1** (Semántica estática). Definimos el siguiente juicio para el sistemas de tipos de MinHaskell:

$$\Gamma \vdash t : T$$

El cual se lee como: *la expresión t tiene tipo T bajo el contexto Γ*. En donde  $\Gamma$  es un conjunto de asignaciones de tipos a variables de la forma  $\{x_1 : T_1 \dots x_n : T_n\}^a$

**Variables**

$$\overline{\Gamma, x : T \vdash x : T}$$



## Valores numéricos

$$\frac{}{\Gamma \vdash \text{num}[n] : \mathbf{Nat}}$$

## Valores Booleanos

$$\frac{}{\Gamma \vdash \text{bool}[\text{False}] : \mathbf{Bool}}$$

$$\frac{}{\Gamma \vdash \text{bool}[\text{True}] : \mathbf{Bool}}$$

## Operadores

$$\frac{\Gamma \vdash t_1 : \mathbf{nat} \quad \Gamma \vdash t_2 : \mathbf{Nat}}{\Gamma \vdash \text{sum}(t_1, t_2) : \mathbf{Nat}}$$

$$\frac{\Gamma \vdash t_1 : \mathbf{nat} \quad \Gamma \vdash t_2 : \mathbf{Nat}}{\Gamma \vdash \text{prod}(t_1, t_2) : \mathbf{Nat}}$$

$$\frac{\Gamma \vdash t_1 : \mathbf{nat} \quad \Gamma \vdash t_2 : \mathbf{Nat}}{\Gamma \vdash \text{sub}(t_1, t_2) : \mathbf{Nat}}$$

$$\frac{\Gamma \vdash t_1 : \mathbf{nat} \quad \Gamma \vdash t_2 : \mathbf{Nat}}{\Gamma \vdash \text{ig}(t_1, t_2) : \mathbf{Bool}}$$

$$\frac{\Gamma \vdash t_1 : \mathbf{nat} \quad \Gamma \vdash t_2 : \mathbf{Nat}}{\Gamma \vdash \text{gt}(t_1, t_2) : \mathbf{Bool}}$$

$$\frac{\Gamma \vdash t_1 : \mathbf{nat} \quad \Gamma \vdash t_2 : \mathbf{Nat}}{\Gamma \vdash \text{lt}(t_1, t_2) : \mathbf{Bool}}$$

## Condicional

$$\frac{\Gamma \vdash t_c : \mathbf{Bool} \quad \Gamma \vdash t_t : T \quad \Gamma \vdash t_e : T}{\Gamma \vdash \text{if}(t_c, t_t, t_e) : T}$$

## Asignaciones Locales

$$\frac{\Gamma \vdash t_v : T \quad \Gamma, x : T \vdash t_b : St}{\Gamma \vdash \text{let}(t_v, x.t_b) : St}$$

## Funciones

$$\frac{\Gamma, x : T \vdash t : St}{\Gamma \vdash \text{fun}(T, x.t) : T \rightarrow St} \quad \frac{\Gamma \vdash f : T \rightarrow St, x : T \vdash t : S}{\Gamma \vdash \text{recfun}(T \rightarrow S, f.x.t) : T \rightarrow S}$$

## Aplicación de función

$$\frac{\Gamma \vdash t_f : T \rightarrow St \quad \Gamma \vdash t_p : T}{\Gamma \vdash \text{app}(t_f, t_p) : St}$$

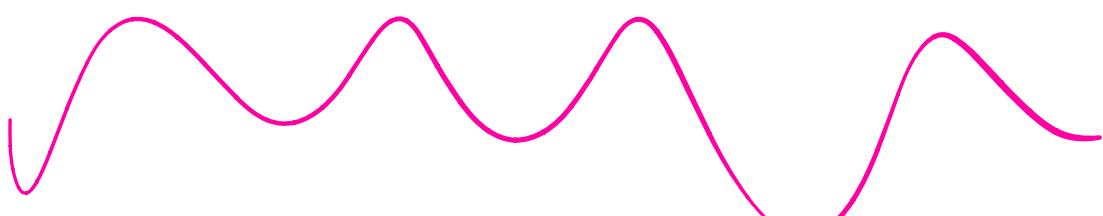
## Operador de punto fijo

$$\frac{\Gamma, x : T \vdash t : T}{\Gamma \vdash \text{fix}(T, x.t) : T}$$

Obsérvese que en el caso de se está asumiendo el mismo tipo que se debe concluir.

---

<sup>a</sup>Se utiliza la notación  $\Gamma, x : T$  para indicar el conjunto  $\Gamma \cup \{x : T\}$



**Ejercicio 2.1.** Para cada expresión de *MinHaskell* enlistada a continuación obtén su representación en **sintaxis abstracta** y aplica las reglas de la **semántica estática** para hacer el análisis de tipos de la expresión.

Representación en sintaxis concreta:

$$1 + (7 - 1)$$

descripción

Representación en sintaxis abstracta:

$$\text{sum}(\text{num}[1], \text{res}(\text{num}[7], \text{num}[1]))$$

Análisis de tipo aplicando la semántica estática:

$$\frac{\frac{\frac{}{\vdash \text{num}[7] : \text{Nat}} \quad \frac{}{\vdash \text{num}[1] : \text{Nat}}}{\vdash \text{num}[1] : \text{Nat}} \quad \frac{}{\vdash \text{res}(\text{num}[7], \text{num}[1]) : \text{Nat}}}{\vdash \text{sum}(\text{num}[1], \text{res}(\text{num}[7], \text{num}[1])) : \text{Nat}}$$

Representación en sintaxis concreta:

$$\text{lam } x :: \text{Nat} \rightarrow x > 1$$

Representación en sintaxis abstracta:

$$\text{fun}(\text{Nat}, x.\text{gt}(x, \text{num}[1]))$$

Análisis de tipo aplicando la semántica estática:

$$\frac{\frac{x : \text{Nat} \vdash x : \text{Nat} \quad x : \text{Nat} \vdash \text{num}[1] : \text{Nat}}{x : \text{Nat} \vdash \text{gt}(x, \text{num}[1]) : \text{Bool}}}{\vdash \text{fun}(\text{Nat}, x.\text{gt}(x, \text{num}[1])) : \text{Bool} \rightarrow \text{Nat}}$$

Representación en sintaxis concreta:

$$\text{let } x = 3 \text{ in if } x < 7 \text{ then } x + (7 - x) \text{ else } x \text{ end}$$

Representación en sintaxis abstracta:

$$\text{let}(\text{num}[3], x.\text{if}(\text{lt}(x, \text{num}[7]), \text{num}[7], \text{sum}(x, \text{res}(\text{num}[7], x))))$$

Análisis de tipo aplicando la semántica estática:

$$\frac{\frac{\frac{(A) \dots}{x : \text{Nat} \vdash \text{lt}(x, \text{num}[7]) : \text{Bool}} \quad \frac{(B) \dots}{x : \text{Nat} \vdash \text{num}[7] : \text{Nat}} \quad \frac{}{x : \text{Nat} \vdash \text{sum}(x, \text{res}(\text{num}[7], x)) : \text{Nat}}}{x : \text{Nat} \vdash \text{if}(\text{lt}(x, \text{num}[7]), \text{num}[7], \text{sum}(x, \text{res}(\text{num}[7], x)))}}{\vdash \text{let}(\text{num}[3], x.\text{if}(\text{lt}(x, \text{num}[7]), \text{num}[7], \text{sum}(x, \text{res}(\text{num}[7], x)))) : \text{Nat}}$$

(A) Análisis semántico para lt

$$\frac{x : \text{Nat} \vdash x : \text{Nat} \quad x : \text{Nat} \vdash \text{num}[7] : \text{Nat}}{x : \text{Nat} \vdash \text{lt}(x, \text{num}[7]) : \text{Bool}}$$

(B) Análisis semántico para sum

$$\frac{\frac{x : \text{Nat} \vdash \text{num}[7] : \text{Nat} \quad x : \text{Nat} \vdash x : \text{Nat}}{x : \text{Nat} \vdash \text{res}(\text{num}[7], x) : \text{Nat}}}{x : \text{Nat} \vdash \text{sum}(x, \text{res}(\text{num}[7], x)) : \text{Nat}}$$

## 2.3 Semántica dinámica

Para concluir con la definición de *MinHaskell* enunciaremos las reglas de la **Semántica Operacional**. Es importante observar que la evaluación perezosa característica de Haskell se debe particularmente a los operadores **app** y **let** dado que la sustitución se hace sin evaluar la expresión que dará el valor en la variable ligada.

- Definición 2.2** (Semántica Operacional de paso pequeño perezosa).
- Conjunto de estados  $S = \{a \mid a \text{ asa}\}$
  - Estados Iniciales  $I = \{a \mid a \text{ asa}, \emptyset \sim a\}$
  - Estados Finales  $F = \{\text{num}[n], \text{bool}[true], \text{bool}[false], \text{lam}(x.t)\}$
  - Transiciones, dadas por las siguientes reglas:

### Condicional

$$\frac{}{if(bool[true], a_t, a_e) \rightarrow a_t} \text{ ifT} \quad \frac{}{if(bool[false], a_t, a_e) \rightarrow a_e} \text{ ifF}$$

$$\frac{a_c \rightarrow a'_c}{if(a_c, a_t, a_e) \rightarrow if(a'_c, a_t, a_e)} \text{ if}$$

### Asignaciones locales

$$\frac{}{let(a_1, x.a_2) \rightarrow a_2[x := a_1]} \text{ let}$$

### Aplicación de función

$$\frac{}{app(lam(x.ab), a_p) \rightarrow ab[x := a_p]} \text{ app}$$

$$\frac{a_f \rightarrow a'_f}{app(a_f, a_p) \rightarrow app(a'_f, a_p)} \text{ appL}$$

$$\frac{}{app(recfun(\mathbf{T}, x.ab), a_p) \rightarrow ab[f := fix(\mathbf{T}, f.x.ab), x := a_p])} \text{ appR}$$

$$\frac{}{app(fix(\mathbf{T}, f.x.ab), a_p) \rightarrow ab[f := fix(\mathbf{T}, f.x.ab), x := a_p]} \text{ appF}$$

### Operador de punto fijo

$$\frac{}{fix(\mathbf{T}, f.a) \rightarrow a[f := fix(\mathbf{T}, f.a)]} \text{ fix}$$

### Operadores

Para acotar el listado de reglas se omite la representación para los operadores cuya definición es la misma a la que se dió para  $EAB$  en el **Capítulo 4: Semántica** definición 2.3, las reglas para operadores booleanos de comparación  $lt$ ,  $<$ ,  $>$ ,  $=$  se define de manera análoga.

### 3 Propiedades de MinHaskell

Para concluir este capítulo revisaremos brevemente las propiedades que la definición de *MinHaskell* posee. Mas adelante revisitaremos esta sección para discutir el resto de éllas enunciando únicamente las propiedades **Seguridad** y **No terminación** del lenguaje.

#### 3.1 Seguridad del lenguaje

Esta propiedad es una característica que nos ayuda a verificar la correctud del sistema, y se interpreta como: “Un programa que cumpla con las restricciones dadas por el sistema de tipos no puede tener una evaluación errónea”. Ligando las definiciones para la **Semántica estática** conciernete al tipado de las expresiones y la **Semántica Dinámica** concerniente a la evaluación de las mismas.

Ésta propiedad posee dos características que se aplican a las expresiones de *MinHaskell*:

- **Progreso**

Engloba la propiedad: “Un programa que cumple con las restricciones de tipado no se bloquea hasta llegar a un valor.” Éste comportamiento está capturado en la definición de progreso enlistada a continuación para las expresiones de *MinHaskell*:

**Definición 3.1** (Propiedad de progreso). Si  $\vdash t : \mathbf{T}$  para algún tipo **T** entonces se cumple una de los siguientes dos casos:

- (A)  $t$  es un valor
- (B) Existe una expresión  $t'$  tal que  $t \rightarrow t'$

- **Preservación**

Referente a la idea: “Un programa que cumple con las restricciones de tipado dará como resultado de la evaluación una expresión correctamente tipada y que potencialmente preserva su mismo tipo”

Ésto se puede observar en las siguientes dos propiedades definidas para las expresiones de *MinHaskell*:

**Definición 3.2** (Unicidad de tipado). Para cualesquiera  $\Gamma$  y expresión  $t$  de **MinHs** existe a lo más un tipo **T** de tal forma que se cumple:  $\Gamma \vdash t : \mathbf{T}$

**Definición 3.3** (Preservación de tipos). Si  $\Gamma \vdash t : \mathbf{T}$  y  $t \rightarrow t'$  entonces  $\Gamma \vdash t' : \mathbf{T}$

## 3.2 No terminación

*MinHaskell* hereda un problema similar a lo que sucedió en el **Cálculo Lambda** con la introducción de los combinadores para implementar la **recursión general**. Éste problema es la existencia de expresiones del lenguaje sintácticamente y semánticamente correctas que producen un estado conocido '**loop**' ó '**estado de ciclado**'

**Ejercicio 3.1.** Demuestra o da un contraejemplo de la propiedad de ~~no~~ terminación para *MinHaskell*

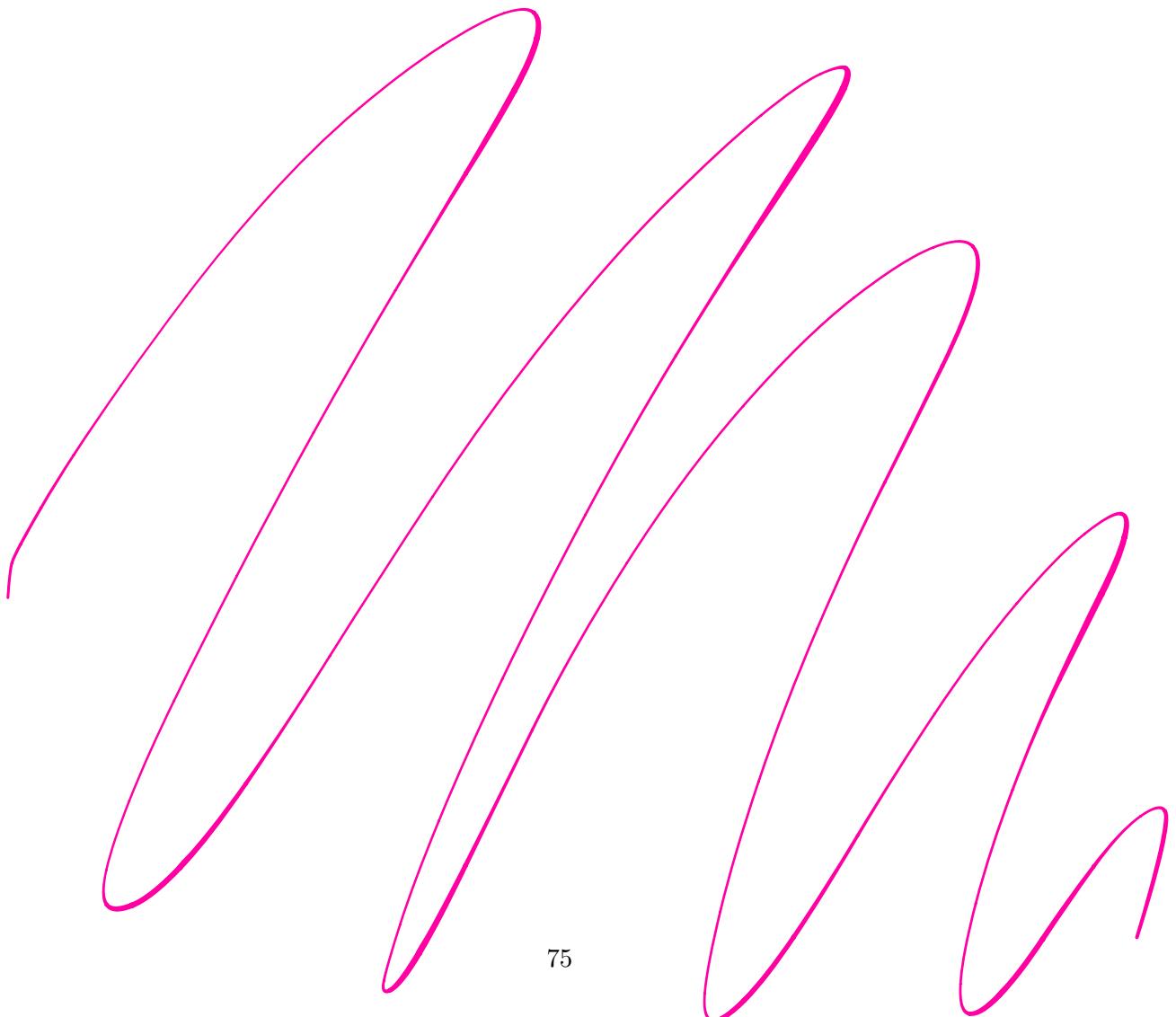
Considera la función identidad que recibe un parámetro y regresa el mismo representada como **x.x** si utilizamos esta expresión como el cuerpo de nuestro operador **fix** obtenemos la expresión:

$$\text{fix}(\text{x}.x) \rightarrow x[x := \text{fix}(\text{x}.x)] = \text{fix}(\text{x}.x) \rightarrow \dots^a$$

Por lo tanto *MinHaskell* **no posee** la propiedad de terminación.

---

<sup>a</sup>Ejemplo extraido de Javier E. Mendoza, Lenguajes de Programación 2023-1 Nota de clase 6 **MinHaskell**, Universidad Nacional Autónoma de México. pp 8



## 4 Ejercicios para el lector

Para concluir el capítulo se enlistan a continuación los ejercicios de comprensión para el lector, similares a los revisados durante el capítulo.

Estos ejercicios pretenden aplicar los diferentes niveles de la semántica y la sintaxis de *MinHaskell* agregando instrucciones para el lenguaje o verificando el tipado sobre la definición de funciones y expresiones<sup>2</sup>.

**Ejercicio 4.1.** Queremos extender la definición de *MinHaskell* para agregar el tipo de dato algebráico *tupla* representado como **Pair(x,y)** (donde ambos elementos no necesariamente tienen que tener el mismo tipo). Así como las proyecciones **Fst** y **Snd**.

Con la descripción anterior responde los siguientes puntos:

- Define la sintaxis concreta para las tuplas.
- Define la regla de sintaxis abstracta para las tuplas.
- Define la regla de la semántica estática para las tuplas.
- Define la(s) reglas de evaluación para las tuplas.
- Cuál es la diferencia entre una expresión que construye la tupla y la proyección que obtiene el elemento? (**Hint:** Las reglas de la semántica dinámica permiten comparar ambos elementos para responder este punto).

**Ejercicio 4.2.** Para las siguientes expresiones de *MinHaskell* determina el tipo y verifica tu respuesta utilizando las reglas de la **Semántica estática**.

- **Pair(3, True)**
- **Snd Pair(3, True)**
- **app(lam(Nat, x.suc(x)), 0)**
- **recfun fact :: (Nat → Nat) x → if IsZero(x) then 1 else n \* fact (n – 1)**
- **let f :: (Nat → Nat) in lam x :: Nat → f x end**

**Ejercicio 4.3.** El operador **Case x of x.g → c1 ; x.g → c2 ; ... end** es la generalización de múltiples expresiones **if else** para evaluar el argumento de entrada o *guardia* (*g*) y el caso al que este es aplicado (*e<sub>n</sub>*) de tal forma que se toma como resultado de la expresión la primera guardia que se evalúe a verdadero de izquierda a derecha (o de arriba hacia abajo como generalmente se escribe el operador **Case**).

Adicionalmente queremos definir una expresión que por omisión sea el resultado del **Case** en caso de que ninguna de las condicionales se evalúe a verdadero:

**Case x of x.g → c1 ; x.g → c2 ; ... ; e<sub>d</sub> end**

<sup>2</sup>Ejercicios 4.3, 4.5 y 4.6 extraídos de Javier E. Mendoza, Kevin P. Ramos, Lenguajes de Programación 2023-1; Boletín de ejercicios 4, Noviembre 2022. Extendiendo la definición del operador **Case** para incluir el caso por omisión.

Con la especificación dada anterior contesta los siguientes incisos:

- Extiende la sintáxis concreta del lenguaje para el operador **Case** (**Hint:** se puede añadir un nivel extra para las expresiones guardadas y la expresión **default**).
- Extiende la sintáxis abstracta del lenguaje para el operador **Case**.
- Extiende la semántica estática del lenguaje para el operador **Case**.
- Extiende la semántica dinámica del lenguaje para el operador **Case** (recuerda que se tiene que seguir la implementación de evaluación perezosa).

**Ejercicio 4.4.** Dada la siguiente expresión **Case**:

**Case**  $n$  **of**  $n < 0 \rightarrow \text{True} ; n = 0 \rightarrow \text{True} ; \text{False}$

- Evalúala utilizando las reglas de semántica dinámica definidas en el inciso anterior.
- Sí  $\Gamma = \{n : \text{Nat}\}$  utiliza las reglas de semántica estática definidas en el inciso anterior para obtener el tipado de la expresión.

**Ejercicio 4.5.** Define las siguientes funciones utilizando la sintáxis concreta de *MinHaskell* y aplica las reglas de la semántica estática para verificar el tipado de las mismas.

- Define la función **exp**( $n_1\ n_2$ ) que eleva el primer argumento a la potencia del segundo.
- Define la función **fibonacci**( $n$ ) que obtiene el  $n$ -ésimo elemento de la sucesión de Fibonacci.
- Evalúa la expresión **exp**(2,2) y **fibonacci**(3).

**Ejercicio 4.6.** Para las siguientes expresiones, verifique el tipado mostrando la derivación mediante la semántica estática.

- $(\text{fun } x : \text{Nat} \rightarrow \text{Nat} \rightarrow (\text{fun } w : \text{Nat} \rightarrow (x\ w) + (x\ w)))\ (\text{fun } y : \text{Nat} \rightarrow y + 1)$
- $(\text{let neg} = (\text{fun } x : \text{Bool} \rightarrow \text{if } x \text{ then false else true}) \text{ in neg}(3 \leq 2) \text{ end})$

## Capítulo 7

# Inferencia de tipos



Una característica interesante de **Haskell** (y deseable para **MinHaskell**) es el mecanismo de **Inferencia de Tipos**. Este mecanismo provee una capa ~~extra que~~ que permite trabajar con expresiones aún si estas no poseen un tipado explícito en cada uno de sus argumentos y valores de retorno.

Para ésto, vamos a revisar los mecanismos para formular las restricciones, la estandarización de variables y aplicación de la inferencia de tipos mediante el algoritmo de unificación para las expresiones de MinHaskell<sup>1</sup>.

*Hay que volver a  
escribir esto*

<sup>1</sup>La definiciones que revisaremos en este capítulo fueron extraídas de: Enríquez Mendoza J., Lenguajes de Programación Nota de clase: Inferencia de tipos. Universidad Nacional Autónoma de México, 2022.

Si es, el problema es que depende de las anotaciones de tipos

Anteriormente se definió mediante la semántica estática las reglas para obtener el tipo de una expresión, no obstante esta semántica no es suficiente para descartar expresiones del lenguaje bien formadas pero cuya evaluación se verá detenida debido a la inconsistencia de tipos.

Esta nueva capa aporta a que al lenguaje se vuelva robusto y constituye una herramienta valiosa para la **seguridad** del mismo, es decir que si una expresión de **MinHaskell** está bien formada y ha sido tipada siguiendo las diferentes reglas y etapas correctamente entonces nuestro programa nos regresará un valor.

Objetivo

No necesariamente.

El interés de este capítulo es la definición de nuestro sistema de inferencia que nos permita tipar las expresiones aplicando diferentes etapas. Para ésto, el objetivo principal será la definición del algoritmo de unificación  $\mu$ .

Planteamiento

El capítulo nos provee de diferentes algoritmos y procedimientos que se aplican a las expresiones de **MinHaskell** para poder asignarle un tipo.

El primer mecanismo de filtrado que se aplica será la **Estandarización de Variables** para obtener  $\alpha$ -equivalencias entre las variables ligadas repetidas dentro de una expresión.

Posteriormente vamos a revisar los juicios para generar las **Restricciones de Tipo** que cada expresión genera para finalmente definir el algoritmo unificador  $\mu$ .

## 1 Estandarización de variables

La **Semántica Estática** de **MinHaskell** está definida para ir agregando al contexto  $\Gamma$  las variables que se van encontrando en las expresiones para que sea consistente con el tipo de los valores que reciben y el tipo de los valores que regresan una vez concluida la evaluación.

El contexto  $\Gamma$  solo puede tener una sola aparición por variable, es decir, nunca podremos tener  $\Gamma = \{x : Nat, x : Bool\}$  dado que  $\Gamma$  es un conjunto de variables y tipos, tener la misma variable con dos tipos distintos es una inconsistencia dentro de nuestro programa y constituye un error en el tipado de la expresión.

Para evitar esta situación se define el proceso conocido como **Estandarización de Variables** que consiste en brindar una  $\alpha$ -equivalencia cuando dos variables ligadas tengan el mismo nombre.

Tomemos como ejemplo la siguiente expresión:

let  $x = true$  in  $x !=$  let  $x = 5$  in  $x \leq x$  end end

No existe <sup>70</sup> este operador

En este caso la expresión está bien formada, no hay ninguna regla de la sintaxis concreta que esté mal aplicada para obtener nuestra expresión, la evaluación tampoco comprende un problema dado que la expresión `let` mas interna se evalúa a `true` y luego se prosigue con la comparación de dos booleanos, sin embargo esta expresión nos genera el contexto  $\Gamma = \{x : Nat, x : Bool\}$  el cual es inconsistente.

El problema puede ser corregido utilizando una  $\alpha - equivalencia$  y renombrando las variables `x` por `x0` y `x1` respectivamente:

```
let x0 = true in x0 != let x1 = 5 in x1 ≤ x1 end end
```

Donde el contexto obtenido será:  $\Gamma = \{x_1 : Nat, x_0 : Bool\}$

**Ejercicio 1.1.** Para la siguiente expresión de **MinHaskell**, expresando el contexto:

```
let x = False in x == ( let x = 6 in (x + x + x) ≤ 10 end ) end
```

$$\Gamma = \{x : Nat, x : Bool\}$$

Aplica la estandarización de variables

```
let x0 = False in x0 == ( let x1 = 6 in (x1 + x1 + x1) ≤ 10 end ) end
```

$$\Gamma = \{x_1 : Nat, x_0 : Bool\}$$

**Ejercicio 1.2.** Para la siguiente expresión de **MinHaskell**, expresando el contexto:

```
let x = True in x or ( let x = 5 in x == 5 end ) end
```

$$\Gamma = \{x : Bool, x : Nat\}$$

Aplica la estandarización de variables

```
let x0 = True in x0 or ( let x1 = 5 in x1 == 5 end ) end
```

$$\Gamma = \{x_1 : Bool, x_0 : Nat\}$$

**Ejercicio 1.3.** Para la siguiente expresión de **MinHaskell**, expresando el contexto:

```
let x = 1 in ( let x = False in ( let x = lam y :: Bool → y or x end ) end ) end
```

$$\Gamma = \{x : Int, x : Bool, x : Bool \rightarrow Bool\}$$

Aplica la estandarización de variables

```
let x0 = 1 in ( let x1 = False in ( let x2 = lam y :: Bool → y or y end ) end ) end
```

$$\Gamma = \{x_0 : Int, x_1 : Bool, x_2 : Bool \rightarrow Bool\}$$

*ejer<sup>80</sup>cicio : propon una semántica  
está tira que resuelva el  
proceso de estandarización*

## 2 Generación de restricciones

Para poder tipar una expresión bien formada de **MinHaskell** sin utilizar las anotaciones de tipo explicitamente en los argumentos y valores de retorno es necesario definir un proceso que nos ayude a obtener información sobre la estructura que ésta posse.

Esta información se conoce como **restricciones** y nos ayuda a atar las condiciones que se deben cumplir para aplicar la inferencia de tipos.

**Definición 2.1** (Conjunto de restricciones). Nuestro objetivo entonces será partir de una expresión y obtener un conjunto de restricciones, ésto se representa como:

$$e \vdash \mathbb{R}$$

que se lee como: "la expresión  $e$  genera el conjunto  $\mathbb{R}$  de restricciones de tipado".

**Definición 2.2** (Extensión de tipos para la generación de restricciones). Para definir el algoritmo se extiende la categoría de tipos como sigue:

$$T ::= X \mid Nat \mid Bool \mid T_1 \rightarrow T_2 \quad [e]^a \quad \boxed{[e]}$$

en donde  $X$  es una variable de tipo. Estas variables nos ayudan en la definición de programas polimórficos, por ejemplo la función general identidad

$$\text{lam } x \Rightarrow x$$

tiene el tipo  $\mathbb{X} \rightarrow \mathbb{X}$  y esta variable de tipo  $X$  va a tomar su valor hasta que la función sea utilizada, por ejemplo en la aplicación

$$(funt x \Rightarrow x) 5$$

la variable  $X$  toma el valor de **Nat**.

*que no  
se  
not  
al pie*

"La expresión  $[e]$  es una construcción sintáctica para definir el tipo de una expresión  $e$  del lenguaje y se lee como: ".el tipo de  $e$ ". Es importante aclarar que los tipos de la forma  $[e]$  no pueden figurar en el tipo resultante del algoritmo de inferencia, el tipo de una expresión debe construirse únicamente con el resto de los tipos.

**Definición 2.3** (Algoritmo de generación de restricciones). Una restricción es una ecuación de la forma  $T_1 = T_2$  en donde  $T_1$  y  $T_2$  son tipos. La ecuación indica que  $T_1$  debe ser igual a  $T_2$  bajo unificación.

### Variables

$$\overline{x_i \mapsto [x_i] = \mathbb{X}_i}$$

Para el tipo de las variables se usará una variable de tipo con el mismo nombre de la variable. Todas las apariciones de la misma variable generarán la misma

restricción y como los nombres de variables son únicos no habrá dos variables distintas con el mismo tipo.

### Valores numéricos

$$\frac{}{num[n] \mapsto [num[n]] = Nat}$$

### Valores Booleanos

$$\frac{}{Bool[b] \mapsto [Bool[b]] = Bool}$$

### Operadores

$$\frac{e_1 \mapsto R_1 \quad e_2 \mapsto R_2}{sum(e_1, e_2) \mapsto R_1, R_2, [e_1] = Nat, [e_2] = Nat, [sum(e_1, e_2)] = Nat}$$

$$\frac{e_1 \mapsto R_1 \quad e_2 \mapsto R_2}{prod(e_1, e_2) \mapsto R_1, R_2, [e_1] = Nat, [e_2] = Nat, [prod(e_1, e_2)] = Nat}$$

$$\frac{e_1 \mapsto R_1 \quad e_2 \mapsto R_2}{sub(e_1, e_2) \mapsto R_1, R_2, [e_1] = Nat, [e_2] = Nat, [sub(e_1, e_2)] = Nat}$$

$$\frac{e_1 \mapsto R_1 \quad e_2 \mapsto R_2}{ig(e_1, e_2) \mapsto R_1, R_2, [e_1] = Nat, [e_2] = Nat, [ig(e_1, e_2)] = Bool}$$

$$\frac{e_1 \mapsto R_1 \quad e_2 \mapsto R_2}{gt(e_1, e_2) \mapsto R_1, R_2, [e_1] = Nat, [e_2] = Nat, [gt(e_1, e_2)] = Bool}$$

$$\frac{e_1 \mapsto R_1 \quad e_2 \mapsto R_2}{lt(e_1, e_2) \mapsto R_1, R_2, [e_1] = Nat, [e_2] = Nat, [lt(e_1, e_2)] = Bool}$$

### Condicional

$$\frac{c \mapsto R_1 \quad t \mapsto R_2 \quad e \mapsto R_3}{if(c, t, e) \mapsto R_1, R_2, R_3, [c] = Bool, [t] = [e], [if(c, t, e)] = [e], [if(c, t, e)] = [t]}$$

### Asignaciones Locales

$$\frac{v \mapsto R_1 \quad b \mapsto R_2}{let(v, x_i.b) \mapsto R_1, R_2, X_i = [v], [let(v, x_i.b)] = [b]}$$

$$\frac{v \mapsto R_1 \quad b \mapsto R_2}{letrec(v, x_i.b) \mapsto R_1, R_2, X_i = [v], [letrec(v, x_i.b)] = [b]}$$

### Funciones

$$\frac{t \mapsto R}{funt(x_i.t) \mapsto R, [funt(x_i.t)] = X_i \rightarrow [t]}$$

### Aplicación de función

$$\frac{f \mapsto R_1 \quad p \mapsto R_2}{app(f, p) \mapsto R_1, R_2, [f] = [p] \rightarrow [app(f, p)]}$$

### 3 Algoritmo de unificación

Una vez obtenida la lista de restricciones asociadas a una expresión  $e$  tenemos toda la información que necesitamos acerca de su estructura para comenzar a unificar los tipos aplicando las restricciones hasta encontrar el tipo más general de la expresión, ó hasta encontrar un error de tipado al asignar dos tipos distintos a la misma expresión.

Para ésto construiremos una composición de sustituciones tomando cada una de las restricciones y sustituyendo el tipo al cuál la expresión está ligada en dicha restricción. Al final se obtendrá la lista de sustituciones necesarias para hallar el tipo más general el cuál será la cabeza de la lista, en caso contrario quiere decir que la unificación falló.

**Definición 3.1** (Algoritmo de unificación). La entrada del algoritmo es una lista de restricciones y la salida es un unificador  $\mu$  en caso de que las restricciones se puedan resolver o fail en caso contrario.

$$\begin{aligned} U([]) &= [] \\ U(T = T : R) &= U(R) \\ U(X = T : R) &= U(R[X := T]) \circ [X := T] && \text{si } X \notin (T) \\ U(X = T : R) &= \text{fail} && \text{si } X \in (T) \\ U(e = T : R) &= U(R[e := T]) \circ [e := T] \\ U(T = X : R) &= U(X = T : R) \\ U(T = e : R) &= U(e = T : R) \\ U(St_1 \rightarrow St_2 = T_1 \rightarrow T_2 : R) &= U(St_1 = T_1 : St_2 = T_2 : R) \\ U(R) &= \text{fail} \end{aligned}$$

En donde  $T$  representa a cualquier tipo y  $X$  a una variable de tipo.

### 4 Algoritmo de inferencia de tipos

**Definición 4.1** (Algoritmo de Inferencia de tipos). Se define el algoritmo  $\mathcal{T}(e)$  de inferencia de tipos que recibe una expresión  $e$  de **minhs** y regresa el tipo de esta expresión. El algoritmo se define con los siguientes pasos:

- Se encuentra la expresión  $e'$  con nombres de variables únicas.
- Se encuentra el conjunto de restricciones  $\mathbb{R}$  tal que  $e' \mapsto \mathbb{R}$ .
- Utilizando la función  $U$  se calcula el unificador mas general  $\mu$  del conjunto de restricciones  $\mathbb{R}$ , tal que  $U(\mathbb{R}) = \mu$ .
- Se busca en  $\mu$  la ecuación  $e' := T$ .
- $T$  es el tipo mas general de  $e$ , es decir,  $\mathcal{T}(e) = T$ .

**Ejercicio 4.1.** Vamos a encontrar el tipo de la expresión:

```
let x = 0 in
  let y = 1 in
    x == y
  end
end
```

que corresponde al árbol de sintaxis abstracta:

$$\text{let}(0, x.\text{let}(1, y.\text{eq}(x, y)))$$

### Renombramiento de variables

$$\text{let}(0, x_0.\text{let}(1, x_1.\text{eq}(x_0, x_1)))$$

### Generación de Restricciones .

- $0 \mapsto [1] = \text{Nat}$
- $1 \mapsto [1] = \text{Nat}$
- $x_0 \mapsto [x_0] = \mathbb{X}_0$
- $x_1 \mapsto [x_1] = \mathbb{X}_1$
- $\text{eq}(x_0, x_1) \mapsto \underbrace{[x_0] = \mathbb{X}_0, [x_1] = \mathbb{X}_1, [x_0] = \text{Nat}, [x_1] = \text{Nat}}_{\mathbb{R}_1}, [\text{eq}(x_0, x_1)] = \text{Bool}$
- $\text{let}(1, x_1.\text{eq}(x_0, x_1)) \mapsto [1] = \text{Nat}, R_1, \mathbb{X}_1 = [1], [\text{let}(1, x_1.\text{eq}(x_0, x_1))] = [\text{eq}(x_0, x_1)]$
- $\text{let}(0, x_0.\text{let}(1, x_1.\text{eq}(x_0, x_1))) \mapsto \underbrace{[0] = \text{Nat}, \mathbb{R}_2, \mathbb{X}_0 = [0]}_{\mathbb{R}_3}, [\text{let}(0, x_0.\text{let}(1, x_1.\text{eq}(x_0, x_1)))] = [\text{let}(1, x_1.\text{eq}(x_0, x_1))]$

Como resultado se obtiene la lista de restricciones  $\mathbb{R}$ :

$$\begin{aligned} \mathbb{R} = & [0] = \text{Nat}, \\ & [1] = \text{Nat} \\ & [x_0] = \mathbb{X}_0 \\ & [x_1] = \mathbb{X}_1 \\ & [x_0] = \text{Nat} \\ & [\text{eq}(x_0, x_1)] = \text{Bool} \\ & \mathbb{X}_1 = [1] \\ & [\text{let}(1, x_1.\text{eq}(x_0, x_1))] = [\text{eq}(x_0, x_1)] \\ & [x_1] = \mathbb{X}_1 \\ & \mathbb{X}_0 = [0] \\ & [\text{let}(0, x_0.\text{let}(1, x_1.\text{eq}(x_0, x_1)))] = [\text{let}(1, x_1.\text{eq}(x_0, x_1))] \end{aligned}$$

### Unificación de $\mathbb{R}$ .

Restricciones	Unificador $\mu$
$[0] = Nat$ $[1] = Nat$ $[x_0] = \mathbb{X}_0$ $[x_1] = \mathbb{X}_1$ $[x_0] = Nat$ $[eq(x_0, x_1)] = Bool$ $\mathbb{X}_1 = [1]$ $[let(1, x_1.eq(x_0, x_1))] = [eq(x_0, x_1)]$ $[x_1] = \mathbb{X}_1$ $\mathbb{X}_0 = [0]$ $[let(0, x_0.let(1, x_1.eq(x_0, x_1)))] = [let(1, x_1.eq(x_0, x_1))]$	
$[1] = Nat$ $[x_0] = \mathbb{X}_0$ $[x_1] = \mathbb{X}_1$ $[x_0] = Nat$ $[eq(x_0, x_1)] = Bool$ $\mathbb{X}_1 = [1]$ $[let(1, x_1.eq(x_0, x_1))] = [eq(x_0, x_1)]$ $[x_1] = \mathbb{X}_1$ $\mathbb{X}_0 = Nat$ $[let(0, x_0.let(1, x_1.eq(x_0, x_1)))] = [let(1, x_1.eq(x_0, x_1))]$	$[0] := Nat$
$[x_0] = \mathbb{X}_0$ $[x_1] = \mathbb{X}_1$ $[x_0] = Nat$ $[eq(x_0, x_1)] = Bool$ $\mathbb{X}_1 = Nat$ $[let(1, x_1.eq(x_0, x_1))] = [eq(x_0, x_1)]$ $[x_1] = \mathbb{X}_1$ $\mathbb{X}_0 = Nat$ $[let(0, x_0.let(1, x_1.eq(x_0, x_1)))] = [let(1, x_1.eq(x_0, x_1))]$	$[0] := Nat$ $[1] := Nat$
$[x_1] = \mathbb{X}_1$ $\mathbb{X}_0 = Nat$ $[eq(x_0, x_1)] = Bool$ $\mathbb{X}_1 = Nat$ $[let(1, x_1.eq(x_0, x_1))] = [eq(x_0, x_1)]$	$[0] := Nat$ $[1] := Nat$ $[x_0] := \mathbb{X}_0$

$[x_1] = \mathbb{X}_1$	
$\mathbb{X}_0 = Nat$	
$[let(0, x_0.let(1, x_1.eq(x_0, x_1)))] = [let(1, x_1.eq(x_0, x_1))]$	
$\mathbb{X}_0 = Nat$	$[0] := Nat$
$[eq(x_0, x_1)] = Bool$	$[1] := Nat$
$\mathbb{X}_1 = Nat$	$[x_0] := \mathbb{X}_0$
$[let(1, x_1.eq(x_0, x_1)))] = [eq(x_0, x_1)]$	$[x_1] := \mathbb{X}_1$
$\mathbb{X}_1 = \mathbb{X}_1$	
$\mathbb{X}_0 = Nat$	
$[let(0, x_0.let(1, x_1.eq(x_0, x_1)))] = [let(1, x_1.eq(x_0, x_1))]$	
$[eq(x_0, x_1)] = Bool$	$[0] := Nat$
$\mathbb{X}_1 = Nat$	$[1] := Nat$
$[let(1, x_1.eq(x_0, x_1)))] = [eq(x_0, x_1)]$	$[x_0] := \mathbb{X}_0$
$\mathbb{X}_1 = \mathbb{X}_1$	$[x_1] := \mathbb{X}_1$
$Nat = Nat$	$\mathbb{X}_0 := Nat$
$[let(0, x_0.let(1, x_1.eq(x_0, x_1)))] = [let(1, x_1.eq(x_0, x_1))]$	
$[eq(x_0, x_1)] = Bool$	$[0] := Nat$
$\mathbb{X}_1 = Nat$	$[1] := Nat$
$[let(1, x_1.eq(x_0, x_1)))] = [eq(x_0, x_1)]$	$[x_0] := \mathbb{X}_0$
$\mathbb{X}_1 = \mathbb{X}_1$	$[x_1] := \mathbb{X}_1$
$Nat = Nat$	$[x_0] := Nat$
$[let(0, x_0.let(1, x_1.eq(x_0, x_1)))] = [let(1, x_1.eq(x_0, x_1))]$	
$\mathbb{X}_1 = Nat$	$[0] := Nat$
$[let(1, x_1.eq(x_0, x_1)))] = Bool$	$[1] := Nat$
$\mathbb{X}_1 = \mathbb{X}_1$	$[x_0] := \mathbb{X}_0$
$Nat = Nat$	$[x_1] := \mathbb{X}_1$
$[let(0, x_0.let(1, x_1.eq(x_0, x_1)))] = [let(1, x_1.eq(x_0, x_1))]$	$[x_0] := Nat$
	$[eq(x_0, x_1)] = Bool$
$[let(1, x_1.eq(x_0, x_1)))] = Bool$	$[0] := Nat$
$Nat = Nat$	$[1] := Nat$
$Nat = Nat$	$[x_0] := \mathbb{X}_0$
$[let(0, x_0.let(1, x_1.eq(x_0, x_1)))] = [let(1, x_1.eq(x_0, x_1))]$	$[x_1] := \mathbb{X}_1$
	$[x_0] := Nat$
	$[eq(x_0, x_1)] := Bool$
	$\mathbb{X}_1 := Nat$
$Nat = Nat$	$[0] := Nat$

$Nat = Nat$	$[1] := Nat$
$[let(0, x_0.let(1, x_1.eq(x_0, x_1)))] = Bool$	$[x_0] := \mathbb{X}_0$
	$[x_1] := \mathbb{X}_1$
	$[x_0] := Nat$
	$[eq(x_0, x_1)] := Bool$
	$\mathbb{X}_1 := Nat$
	$[let(1, x_1.eq(x_0, x_1))] := Bool$
$[let(0, x_0.let(1, x_1.eq(x_0, x_1)))] = Bool$	$[1] := Nat$
	$[x_0] := \mathbb{X}_0$
	$[x_1] := \mathbb{X}_1$
	$[x_0] := Nat$
	$[eq(x_0, x_1)] := Bool$
	$\mathbb{X}_1 := Nat$
	$[let(1, x_1.eq(x_0, x_1))] := Bool$
$[let(0, x_0.let(1, x_1.eq(x_0, x_1)))] = Bool$	$[0] := Nat$
	$[1] := Nat$
	$[x_0] := \mathbb{X}_0$
	$[x_1] := \mathbb{X}_1$
	$[eq(x_0, x_1)] := Bool$
	$\mathbb{X}_1 := Nat$
	$[let(1, x_1.eq(x_0, x_1))] := Bool$
	$[0] := Nat$
	$[1] := Nat$
	$[x_0] := \mathbb{X}_0$
	$[x_1] := \mathbb{X}_1$
	$[eq(x_0, x_1)] := Bool$
	$\mathbb{X}_1 := Nat$
	$[let(1, x_1.eq(x_0, x_1))] := Bool$
	$[let(0, x_0.let(1, x_1.eq(x_0, x_1)))] := Bool$

De el proceso anterior se puede concluir que el tipo de la expresión es *Bool*

**Ejercicio 4.2.** Vamos a encontrar el tipo de la expresión:

```
(recfun fibonacci n =>
  if (n < 2)
    then 1
  else fibonacci (n - 1) + fibonacci (n - 2) ) 4
```

que corresponde al árbol de sintaxis abstracta:

$$app(recfun(fibonacci.n.if(lt(n, 2), 1, sum(app(fibonacci, (sub(n, 1))), app(fibonacci, (sub(n, 2))))))), 4)$$

### Renombramiento de variables

$$app(recfun(x_0.x_1.if(lt(x_1, 2), 1, sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))))), 4)$$

### Generación de Restricciones .

- $1 \mapsto [1] = Nat$
- $2 \mapsto [2] = Nat$
- $x_0 \mapsto [x_0] = X_0$
- $x_1 \mapsto [x_1] = \mathbb{X}_1$
- $sub(x_1, 1) \mapsto \underbrace{[x_1] = \mathbb{X}_1, [1] = Nat, [x_1] = Nat, [sub(x_1, 1)] = Nat}_{R1}$
- $sub(x_1, 2) \mapsto \underbrace{[x_1] = \mathbb{X}_1, [2] = Nat, [x_1] = Nat, [sub(x_1, 2)] = Nat}_{R2}$
- $app(x_0, sub(x_1, 1)) \mapsto \underbrace{[x_0] = X_0, R_1, [x_0] = [sub(x_1, 1)] \mapsto [app(x_0, sub(x_1, 1))]}_{R3}$
- $app(x_0, sub(x_1, 2)) \mapsto \underbrace{[x_0] = X_0, R_2, [x_0] = [sub(x_1, 2)] \mapsto [app(x_0, sub(x_1, 2))]}_{R4}$
- $sum(app(x_0, sub(x_1, 1)), app(x_0, sub(x_1, 2))) \mapsto \underbrace{R_3, R_4. [app(x_0, sub(x_1, 1))] = Nat, [app(x_0, sub(x_1, 2))] = Nat, [app(x_0, sub(x_1, 2))] = Nat}_{R5}$
- $lt(x_1, 2) \mapsto \underbrace{[x_1] = X_1, [2] = Nat, [x_1] = Nat, [lt(x_1, 2)] = Bool}_{R6}$
- $if(lt(x_1, 2), 1, sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2)))))) \mapsto \underbrace{R_6, [1] = Nat, R_5, [1] = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))], [if(...)] = [1], [if(...)]}_{R7}$
- $recfun(x_0.x_1.if(lt(x_1, 2), 1, sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2)))))) \mapsto \underbrace{R_7, [recfun(...)] = X_1 \mapsto [if(...)]}_{R8}$
- $app(recfun(...), 4) \mapsto \underbrace{R_8, [4] = Nat, [recfun(...)] = [4] \mapsto [app(recfun(...), 4)]}_{R9}$

Como resultado se obtiene la lista de restricciones  $\mathbb{R}$ :

$$\begin{aligned}
\mathbb{R} = & [x_1] = X_1 \\
& [2] = \text{Nat} \\
& [lt(x_1, 2)] = \text{Bool} \\
& [1] = \text{Nat} \\
& [x_0] = X_0 \\
& [x_1] = \mathbb{X}_1 \\
& [x_1] = \text{Nat} \\
& [sub(x_1, 1)] = \text{Nat} \\
& [x_0] = [sub(x_1, 1)] \mapsto [app(x_0, sub(x_1, 1))] \\
& [2] = \text{Nat} \\
& [sub(x_1, 2)] = \text{Nat} \\
& [x_0] = [sub(x_1, 2)] \mapsto [app(x_0, sub(x_1, 2))] \\
& [app(x_0, sub(x_1, 1))] = \text{Nat} \\
& [app(x_0, sub(x_1, 2))] = \text{Nat} \\
& [1] = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))]
\end{aligned}$$

$[if(\dots)] = [1]$   
 $[if(\dots)] = [sum(\dots)]$   
 $[recfun(\dots)] = X_1 \mapsto [if(\dots)]$   
 $[4] = \text{Nat}$   
 $[recfun(\dots)] = [4] \mapsto [app(recfun(\dots), 4)]$

**Unificación de  $\mathbb{R}$ .**

Restricciones	Unificador $\mu$
$[x_1] = X_1$ $[2] = \text{Nat}$ $[lt(x_1, 2)] = \text{Bool}$ $[1] = \text{Nat}$ $[x_0] = X_0$ $[x_1] = \mathbb{X}_1$ $[x_1] = \text{Nat}$ $[sub(x_1, 1)] = \text{Nat}$ $[x_0] = [sub(x_1, 1)] \mapsto [app(x_0, sub(x_1, 1))]$ $[2] = \text{Nat}$ $[sub(x_1, 2)] = \text{Nat}$ $[x_0] = [sub(x_1, 2)] \mapsto [app(x_0, sub(x_1, 2))]$ $[app(x_0, sub(x_1, 1))] = \text{Nat}$ $[app(x_0, sub(x_1, 2))] = \text{Nat}$	

$[1] = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2)))))]$ $[if(...)] = [1]$ $[if(...)] = [sum(...)]$ $[recfun(...)] = X_1 \mapsto [if(...)]$ $[4] = Nat$ $[recfun(...)] = [4] \mapsto [app(recfun(...), 4)]$	
$[2] = Nat$ $[lt(x_1, 2)] = Bool$ $[1] = Nat$ $[x_0] = X_0$ $X_1 = \mathbb{X}_1$ $X_1 = Nat$ $[sub(x_1, 1)] = Nat$ $[x_0] = [sub(x_1, 1)] \mapsto [app(x_0, sub(x_1, 1))]$ $[2] = Nat$ $[sub(x_1, 2)] = Nat$ $[x_0] = [sub(x_1, 2)] \mapsto [app(x_0, sub(x_1, 2))]$ $[app(x_0, sub(x_1, 1))] = Nat$ $[app(x_0, sub(x_1, 2))] = Nat$ $[1] = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2)))))]$ $[if(...)] = [1]$ $[if(...)] = [sum(...)]$ $[recfun(...)] = X_1 \mapsto [if(...)]$ $[4] = Nat$ $[recfun(...)] = [4] \mapsto [app(recfun(...), 4)]$	$[x_1] := X_1$
$[lt(x_1, 2)] = Bool$ $[1] = Nat$ $[x_0] = X_0$ $X_1 = \mathbb{X}_1$ $X_1 = Nat$ $[sub(x_1, 1)] = Nat$ $[x_0] = [sub(x_1, 1)] \mapsto [app(x_0, sub(x_1, 1))]$ $Nat = Nat$ $[sub(x_1, 2)] = Nat$ $[x_0] = [sub(x_1, 2)] \mapsto [app(x_0, sub(x_1, 2))]$ $[app(x_0, sub(x_1, 1))] = Nat$	$[x_1] := X_1$ $[2] := Nat$



$[if(\dots)] = [sum(\dots)]$ $[recfun(\dots)] = X_1 \mapsto [if(\dots)]$ $[4] = Nat$ $[recfun(\dots)] = [4] \mapsto [app(recfun(\dots), 4)]$	
$X_1 = \mathbb{X}_1$ $X_1 = Nat$ $[sub(x_1, 1)] = Nat$ $[x_0] = [sub(x_1, 1)] \mapsto [app(x_0, sub(x_1, 1))]$ $Nat = Nat$ $[sub(x_1, 2)] = Nat$ $X_0 = [sub(x_1, 2)] \mapsto [app(x_0, sub(x_1, 2))]$ $[app(x_0, sub(x_1, 1))] = Nat$ $[app(x_0, sub(x_1, 2))] = Nat$ $Nat = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))]$ $[if(\dots)] = Nat$ $[if(\dots)] = [sum(\dots)]$ $[recfun(\dots)] = X_1 \mapsto [if(\dots)]$ $[4] = Nat$ $[recfun(\dots)] = [4] \mapsto [app(recfun(\dots), 4)]$	$[x_1] := X_1$ $[2] := Nat$ $[lt(x_1, 2)] = Bool$ $[1] = Nat$ $[x_0] = X_0$
$X_1 = Nat$ $[sub(x_1, 1)] = Nat$ $[x_0] = [sub(x_1, 1)] \mapsto [app(x_0, sub(x_1, 1))]$ $Nat = Nat$ $[sub(x_1, 2)] = Nat$ $X_0 = [sub(x_1, 2)] \mapsto [app(x_0, sub(x_1, 2))]$ $[app(x_0, sub(x_1, 1))] = Nat$ $[app(x_0, sub(x_1, 2))] = Nat$ $Nat = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))]$ $[if(\dots)] = Nat$ $[if(\dots)] = [sum(\dots)]$ $[recfun(\dots)] = X_1 \mapsto [if(\dots)]$ $[4] = Nat$ $[recfun(\dots)] = [4] \mapsto [app(recfun(\dots), 4)]$	$[x_1] := X_1$ $[2] := Nat$ $[lt(x_1, 2)] = Bool$ $[1] = Nat$ $[x_0] = X_0$ $X_1 = \mathbb{X}_1$
$X_1 = Nat$ $[sub(x_1, 1)] = Nat$ $[x_0] = [sub(x_1, 1)] \mapsto [app(x_0, sub(x_1, 1))]$ $[lt(x_1, 2)] = Bool$	$[x_1] := X_1$ $[2] := Nat$ $[lt(x_1, 2)] = Bool$

$Nat = Nat$ $[sub(x_1, 2)] = Nat$ $X_0 = [sub(x_1, 2)] \mapsto [app(x_0, sub(x_1, 2))]$ $[app(x_0, sub(x_1, 1))] = Nat$ $[app(x_0, sub(x_1, 2))] = Nat$ $Nat = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2)))))]$ $[if(...)] = Nat$ $[if(...)] = [sum(...)]$ $[recfun(...)] = X_1 \mapsto [if(...)]$ $[4] = Nat$ $[recfun(...)] = [4] \mapsto [app(recfun(...), 4)]$	$[1] = Nat$ $[x_0] = X_0$
$[sub(x_1, 1)] = Nat$ $[x_0] = [sub(x_1, 1)] \mapsto [app(x_0, sub(x_1, 1))]$ $Nat = Nat$ $[sub(x_1, 2)] = Nat$ $X_0 = [sub(x_1, 2)] \mapsto [app(x_0, sub(x_1, 2))]$ $[app(x_0, sub(x_1, 1))] = Nat$ $[app(x_0, sub(x_1, 2))] = Nat$ $Nat = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2)))))]$ $[if(...)] = Nat$ $[if(...)] = [sum(...)]$ $[recfun(...)] = X_1 \mapsto [if(...)]$ $[4] = Nat$ $[recfun(...)] = [4] \mapsto [app(recfun(...), 4)]$	$[x_1] := X_1$ $[2] := Nat$ $[lt(x_1, 2)] = Bool$ $[1] = Nat$ $[x_0] = X_0$ $X_1 = Nat$
$[x_0] = Nat \mapsto [app(x_0, sub(x_1, 1))]$ $Nat = Nat$ $[sub(x_1, 2)] = Nat$ $X_0 = [sub(x_1, 2)] \mapsto [app(x_0, sub(x_1, 2))]$ $[app(x_0, sub(x_1, 1))] = Nat$ $[app(x_0, sub(x_1, 2))] = Nat$ $Nat = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2)))))]$ $[if(...)] = Nat$ $[if(...)] = [sum(...)]$ $[recfun(...)] = X_1 \mapsto [if(...)]$ $[4] = Nat$	$[x_1] := X_1$ $[2] := Nat$ $[lt(x_1, 2)] = Bool$ $[1] = Nat$ $[x_0] = X_0$ $X_1 = Nat$ $[sub(x_1, 1)] = Nat$

$[recfun(\dots)] = [4] \mapsto [app(recfun(\dots), 4)]$	
$Nat = Nat$	$[x_1] := X_1$
$[sub(x_1, 2)] = Nat$	$[2] := Nat$
$X_0 = [sub(x_1, 2)] \mapsto [app(x_0, sub(x_1, 2))]$	$[lt(x_1, 2)] = Bool$
$[app(x_0, sub(x_1, 1))] = Nat$	$[1] = Nat$
$[app(x_0, sub(x_1, 2))] = Nat$	$[x_0] = X_0$
$Nat = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))]$	$X_1 = Nat$
$[if(\dots)] = Nat$	$[sub(x_1, 1)] = Nat$
$[if(\dots)] = [sum(\dots)]$	$[x_0] = Nat \mapsto [app(x_0, sub(x_1, 1))]$
$[recfun(\dots)] = X_1 \mapsto [if(\dots)]$	
$[4] = Nat$	
$[recfun(\dots)] = [4] \mapsto [app(recfun(\dots), 4)]$	
$[sub(x_1, 2)] = Nat$	$[x_1] := X_1$
$X_0 = [sub(x_1, 2)] \mapsto [app(x_0, sub(x_1, 2))]$	$[2] := Nat$
$[app(x_0, sub(x_1, 1))] = Nat$	$[lt(x_1, 2)] := Bool$
$[app(x_0, sub(x_1, 2))] = Nat$	$[1] := Nat$
$Nat = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))]$	$[x_0] := X_0$
$[if(\dots)] = Nat$	$X_1 := Nat$
$[if(\dots)] = [sum(\dots)]$	$[sub(x_1, 1)] := Nat$
$[recfun(\dots)] = X_1 \mapsto [if(\dots)]$	$[x_0] := Nat \mapsto [app(x_0, sub(x_1, 1))]$
$[4] = Nat$	$Nat := Nat$
$[recfun(\dots)] = [4] \mapsto [app(recfun(\dots), 4)]$	
$[sub(x_1, 2)] = Nat$	$[x_1] := X_1$
$X_0 = [sub(x_1, 2)] \mapsto [app(x_0, sub(x_1, 2))]$	$[2] := Nat$
$[app(x_0, sub(x_1, 1))] = Nat$	$[lt(x_1, 2)] := Bool$
$[app(x_0, sub(x_1, 2))] = Nat$	$[1] := Nat$
$Nat = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))]$	$[x_0] := X_0$
$[if(\dots)] = Nat$	$X_1 := Nat$
$[if(\dots)] = [sum(\dots)]$	$[sub(x_1, 1)] := Nat$
$[recfun(\dots)] = X_1 \mapsto [if(\dots)]$	$[x_0] := Nat \mapsto [app(x_0, sub(x_1, 1))]$
$[4] = Nat$	
$[recfun(\dots)] = [4] \mapsto [app(recfun(\dots), 4)]$	
$X_0 = [sub(x_1, 2)] \mapsto [app(x_0, sub(x_1, 2))]$	$[x_1] := X_1$
$[app(x_0, sub(x_1, 1))] = Nat$	$[2] := Nat$
$[app(x_0, sub(x_1, 2))] = Nat$	$[lt(x_1, 2)] := Bool$
$Nat = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))]$	$[1] := Nat$

$[if(...)] = Nat$	$[x_0] := X_0$
$[if(...)] = [sum(...)]$	$X_1 := Nat$
$[recfun(...)] = X_1 \mapsto [if(...)]$	$[sub(x_1, 1)] := Nat$
$[4] = Nat$	$[x_0] := Nat \mapsto [app(x_0, sub(x_1, 1))]$
$[recfun(...)] = [4] \mapsto [app(recfun(...), 4)]$	$[sub(x_1, 2)] = Nat$
$X_0 = Nat \mapsto [app(x_0, sub(x_1, 2))]$	$[x_1] := X_1$
$[app(x_0, sub(x_1, 1))] = Nat$	$[2] := Nat$
$[app(x_0, sub(x_1, 2))] = Nat$	$[lt(x_1, 2)] := Bool$
$Nat = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))]$	$[1] := Nat$
$[if(...)] = Nat$	$[x_0] := X_0$
$[if(...)] = [sum(...)]$	$X_1 := Nat$
$[recfun(...)] = X_1 \mapsto [if(...)]$	$[sub(x_1, 1)] := Nat$
$[4] = Nat$	$[sub(x_1, 2)] = Nat$
$[recfun(...)] = [4] \mapsto [app(recfun(...), 4)]$	
$[x_0] := Nat \mapsto [app(x_0, sub(x_1, 1))]$	
$[app(x_0, sub(x_1, 1))] = Nat$	$[x_1] := X_1$
$[app(x_0, sub(x_1, 2))] = Nat$	$[2] := Nat$
$Nat = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))]$	$[lt(x_1, 2)] := Bool$
$[if(...)] = Nat$	$[1] := Nat$
$[if(...)] = [sum(...)]$	$[x_0] := X_0$
$[recfun(...)] = X_1 \mapsto [if(...)]$	$X_1 := Nat$
$[4] = Nat$	$[sub(x_1, 1)] := Nat$
$[recfun(...)] = [4] \mapsto [app(recfun(...), 4)]$	$[sub(x_1, 2)] = Nat$
	$[x_0] := Nat \mapsto [app(x_0, sub(x_1, 1))]$
$[app(x_0, sub(x_1, 2))] = Nat$	$[x_1] := X_1$
$Nat = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))]$	$[2] := Nat$
$[if(...)] = Nat$	$[lt(x_1, 2)] := Bool$
$[if(...)] = [sum(...)]$	$[1] := Nat$
$[recfun(...)] = X_1 \mapsto [if(...)]$	$[x_0] := X_0$
$[4] = Nat$	$X_1 := Nat$
$[recfun(...)] = [4] \mapsto [app(recfun(...), 4)]$	$[sub(x_1, 1)] := Nat$
	$[sub(x_1, 2)] = Nat$
	$[x_0] := Nat \mapsto [app(x_0, sub(x_1, 1))]$
	$[app(x_0, sub(x_1, 1))] := Nat$
$Nat = [sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))]$	$[x_1] := X_1$
$[if(...)] = Nat$	$[2] := Nat$

$[if(\dots)] = [sum(\dots)]$ $[recfun(\dots)] = X_1 \mapsto [if(\dots)]$ $[4] = Nat$ $[recfun(\dots)] = [4] \mapsto [app(recfun(\dots), 4)]$	$[lt(x_1, 2)] := Bool$ $[1] := Nat$ $[x_0] := X_0$ $X_1 := Nat$ $[sub(x_1, 1)] := Nat$ $[sub(x_1, 2)] := Nat$ $[x_0] := Nat \mapsto [app(x_0, sub(x_1, 1))]$ $[app(x_0, sub(x_1, 1))] := Nat$ $[app(x_0, sub(x_1, 2))] := Nat$
$[if(\dots)] = Nat$ $[if(\dots)] = Nat$ $[recfun(\dots)] = X_1 \mapsto [if(\dots)]$ $[4] = Nat$ $[recfun(\dots)] = [4] \mapsto [app(recfun(\dots), 4)]$	$[x_1] := X_1$ $[2] := Nat$ $[lt(x_1, 2)] := Bool$ $[1] := Nat$ $[x_0] := X_0$ $X_1 := Nat$ $[sub(x_1, 1)] := Nat$ $[sub(x_1, 2)] := Nat$ $[x_0] := Nat \mapsto [app(x_0, sub(x_1, 1))]$ $[app(x_0, sub(x_1, 1))] := Nat$ $[app(x_0, sub(x_1, 2))] := Nat$ $[sum(app(\dots), app(\dots))] := Nat$
$Nat = Nat$ $[recfun(\dots)] = X_1 \mapsto [if(\dots)]$ $[4] = Nat$ $[recfun(\dots)] = [4] \mapsto [app(recfun(\dots), 4)]$	$[x_1] := X_1$ $[2] := Nat$ $[lt(x_1, 2)] := Bool$ $[1] := Nat$ $[x_0] := X_0$ $X_1 := Nat$ $[sub(x_1, 1)] := Nat$ $[sub(x_1, 2)] := Nat$ $[x_0] := Nat \mapsto [app(x_0, sub(x_1, 1))]$ $[app(x_0, sub(x_1, 1))] := Nat$ $[app(x_0, sub(x_1, 2))] := Nat$ $[sum(app(\dots), app(\dots))] := Nat$ $[if(\dots)] := Nat$
$Nat = Nat$ $[recfun(\dots)] = X_1 \mapsto Nat$	$[x_1] := X_1$ $[2] := Nat$

$[4] = Nat$	$[lt(x_1, 2)] := Bool$
$[recfun(\dots)] = [4] \mapsto [app(recfun(\dots), 4)]$	$[1] := Nat$
	$[x_0] := X_0$
	$X_1 := Nat$
	$[sub(x_1, 1)] := Nat$
	$[sub(x_1, 2)] := Nat$
	$[x_0] := Nat \mapsto [app(x_0, sub(x_1, 1))]$
	$[app(x_0, sub(x_1, 1))] := Nat$
	$[app(x_0, sub(x_1, 2))] := Nat$
	$[sum(app(\dots), app(\dots))] := Nat$
	$[if(\dots)] := Nat$
$[recfun(\dots)] = X_1 \mapsto Nat$	$[x_1] := X_1$
$[4] = Nat$	$[2] := Nat$
$[recfun(\dots)] = [4] \mapsto [app(recfun(\dots), 4)]$	$[lt(x_1, 2)] := Bool$
	$[1] := Nat$
	$[x_0] := X_0$
	$X_1 := Nat$
	$[sub(x_1, 1)] := Nat$
	$[sub(x_1, 2)] := Nat$
	$[x_0] := Nat \mapsto [app(x_0, sub(x_1, 1))]$
	$[app(x_0, sub(x_1, 1))] := Nat$
	$[app(x_0, sub(x_1, 2))] := Nat$
	$[sum(app(\dots), app(\dots))] := Nat$
	$[if(\dots)] := Nat$
$[4] = Nat$	$[x_1] := X_1$
$X_1 \mapsto Nat = [4] \mapsto [app(recfun(\dots), 4)]$	$[2] := Nat$
	$[lt(x_1, 2)] := Bool$
	$[1] := Nat$
	$[x_0] := X_0$
	$X_1 := Nat$
	$[sub(x_1, 1)] := Nat$
	$[sub(x_1, 2)] := Nat$
	$[x_0] := Nat \mapsto [app(x_0, sub(x_1, 1))]$
	$[app(x_0, sub(x_1, 1))] := Nat$
	$[app(x_0, sub(x_1, 2))] := Nat$
	$[sum(app(\dots), app(\dots))] := Nat$

	$[if(\dots)] := Nat$ $[recfun(\dots)] = X_1 \mapsto Nat$
$X_1 \mapsto Nat = Nat \mapsto [app(recfun(\dots), 4)]$	$[x_1] := X_1$ $[2] := Nat$ $[lt(x_1, 2)] := Bool$ $[1] := Nat$ $[x_0] := X_0$ $X_1 := Nat$ $[sub(x_1, 1)] := Nat$ $[sub(x_1, 2)] := Nat$ $[x_0] := Nat \mapsto [app(x_0, sub(x_1, 1))]$ $[app(x_0, sub(x_1, 1))] := Nat$ $[app(x_0, sub(x_1, 2))] := Nat$ $[sum(app(\dots), app(\dots))] := Nat$ $[if(\dots)] := Nat$ $[recfun(\dots)] = X_1 \mapsto Nat$ $[4] = Nat$
	$[x_1] := X_1$ $[2] := Nat$ $[lt(x_1, 2)] := Bool$ $[1] := Nat$ $[x_0] := X_0$ $X_1 := Nat$ $[sub(x_1, 1)] := Nat$ $[sub(x_1, 2)] := Nat$ $[x_0] := Nat \mapsto [app(x_0, sub(x_1, 1))]$ $[app(x_0, sub(x_1, 1))] := Nat$ $[app(x_0, sub(x_1, 2))] := Nat$ $[sum(app(\dots), app(\dots))] := Nat$ $[if(\dots)] := Nat$ $[recfun(\dots)] = X_1 \mapsto Nat$ $[4] = Nat$ $X_1 = Nat$ $[app(recfun(\dots), 4)] = Nat$

De el proceso anterior se puede concluir que el tipo de la expresión es *Nat*

## 5 Ejercicios para el lector

**Ejercicio 5.1.** Dada la siguiente expresión de MinHaskell, obtén el tipo más general del unificador  $\mu$  utilizando el algoritmo de inferencia.

```
(recfun factorial n =>
    if (n < 0)
        then 1
    else factorial (n - 1) * n ) 9
```

**Ejercicio 5.2.** Dada la siguiente expresión de MinHaskell, obtén el tipo más general del unificador  $\mu$  utilizando el algoritmo de inferencia<sup>a</sup>

```
fun x -> app(x,x)
```

<sup>a</sup>Ejercicio extraido de: Lenguajes de Programación 2022-1 Nota de clase 8: Inferencia de tipos, Enriquez Mendoza Javier. Universidad Nacional Autónoma de México. Noviembre 2022.

ejer cicio

• ¿Cuál es la complejidad de cada parte del algoritmo?

• ¿Cuál es la complejidad total?

## Capítulo 8

# Máquinas abstractas



Una máquina abstracta define una colección de estados y reglas de transición que nos permiten obtener información acerca de la ejecución de un programa por cada cómputo que este realiza. Este tipo de mecanismos de evaluación ya han sido revisados en cursos previos, específicamente en **Automatas y Lenguajes de programación**, donde se estudian las máquinas de turing, los autómatas y los autómatas de pila que son los modelos mas importantes de este tipo de sistemas.

En este capítulo introduciremos un sistema similar mezclando las ideas para definir un sistema de ejecución para *MinHaskell* conocidas como las máquinas  $\mathcal{H}$  y  $\mathcal{J}$ .

## Objetivo

En este capítulo exploraremos la definición de la máquina  $\mathcal{H}$  definiendo la colección de estados, computos y los axiomas de transición que nos permitan visualizar el proceso de evaluación de un expresión cerrada de  $MinHs$ .

Ejemplos de programas válidos serán revisados para ilustrar los compús. De forma similar la pila de ejecución será introducida para controlar aquellos que se quedan pendientes de evaluación hasta obtener un valor que pueda ser regresado al computo en el tope de la pila para continuar con la ejecución del programa. Posteriormente abstraeremos el manejo de variables para extender la abstracción definiendo así la máquina  $\mathcal{J}$

## Planteamiento

El capítulo está estructurado para presentar los estados y axiomas de reglas de transición para definir las máquinas  $\mathcal{H}$  y  $\mathcal{J}$  seguido de la ejecución detallada paso a paso de programas correctos de  $MinHs$  para ilustrar ambas definiciones.

Finalmente se introducen los conceptos de **Alcance** y **Closure**<sup>1</sup>.

# 1 La Máquina $\mathcal{H}$

Para la implementación de este modelo es necesaria la intrducción de las **marcos** que representan los computos pendientes que encontramos en la expresión de  $MinHs$  que queremos evaluar. Adicionalmente se presenta la **pila de ejecución** para llevar un control de dichos computos que serán utilizados como "placeholders".

Cada marco será empujado al tope de la pila para evaluar las subexpresiones asociadas a él. Una vez concluida la evaluación de las subexpresiones de éste computo, el valor obtenido es empujado a la pila y detenido hasta que todas las subexpresiones hayan sido evaluadas ó es utilizado para ejecutar la instrucción en el tope de la pila (según sea el caso aplicable por axioma).

Esta implementación resulta particularmente cómoda para seguir la ejecución de un programa de una manera simple y amigable sin embargo posteriores optimizaciones serán propuestas con la introducción de la máquina  $\mathcal{J}$

---

<sup>1</sup>Las definiciones de este capítulo fueron extendidas para presentar operadores genéricos para las reglas de transición, el extracto original pertenece a: Enríquez Mendoza J., Lenguajes de Programación Notas de clase: Máquinas Abstractas. Universidad Nacional Autónoma de México, 2022.

## 1.1 Marcos y la pila de control

**Definición 1.1** (Marcos). Los marcos son esqueletos estructurales que registran los cómputos pendientes de una expresión. En nuestra definición, el símbolo  $\square$  indica el lugar en donde se está llevando a cabo la evaluación actual.

**Operadores primitivos**

$$\frac{}{\mathbf{O}(\square, e_2) \text{ marco}} \quad \frac{}{\mathbf{O}(v_1, \square) \text{ marco}}$$

El resto de los operadores binarios son definidos análogamente.

**Condicional**

$$\frac{}{if(\square, e_2, e_3) \text{ marco}}$$

**Aplicación de función**

$$\frac{}{app(\square, e_2) \text{ marco}}$$

**Definición 1.2** (Pila de control). Una pila de control está formada a partir de marcos, y se define inductivamente como sigue:

$$\frac{}{\diamond \text{ pila}} \text{ vacía} \quad \frac{\mathbf{m} \text{ marco} \quad P \text{ pila}}{\mathbf{m}; P \text{ pila}} \text{ top}$$

## 1.2 Estados

**Definición 1.3** (Estados de la máquina  $\mathcal{H}$ ). Los estados están compuestos de una pila de control  $P$  y una expresión  $e$  cerrada y son de alguna de las siguientes formas:

- **Estados de evaluación:** Se evalúa  $e$  siendo  $P$  la pila de control y lo denotamos como  $P \succ e$
- **Estados de retorno:** Devuelve el valor  $v$  a la pila de control  $P$ , que denotamos como  $Pv$

En donde se distinguen dos tipos de estados en particular:

- **Estados iniciales:** comienzan la evaluación con la pila vacía denotados como  $\diamond \succ e$ .
- **Estados finales:** regresan un valor a la pila vacía y se denota  $\diamond v$

### 1.3 Transiciones

**Definición 1.4** (Transiciones para la máquina  $\mathcal{H}$ ). Las transiciones se definen por medio de la relación  $\rightarrow_h$  y es de la forma:

$$P \succ e \rightarrow_h P' \succ e'$$

en donde los símbolos  $\succ$  se pueden sustituir en ambos casos por  $.$

Las reglas para las expresiones de son:

**Valores** Los valores del lenguaje son números, booleanos y funciones y la evaluación de un valor simplemente lo regresa como resultado a la pila, pues un valor ya finalizo su proceso de evaluación.

$$\overline{P \succ v \rightarrow_h P \prec v}$$

**Operaciones** Definimos el proceso para evaluar cualquier operador de la siguiente forma: Para evaluar  $\mathbf{O}(e_1, e_2)$  agregamos el marco  $\mathbf{O}(\square, e_2)$  a la pila y evaluamos  $e_1$ .

$$\overline{P \succ \mathbf{O}(e_1, e_2) \rightarrow_h \mathbf{O}(\square, e_2); P \succ e_1}$$

Si tenemos en el tope de la pila el marco  $\mathbf{O}(\square, e_2)$  y se regresa como resultado un valor  $v$ , entonces, evaluamos  $e_2$  y sustituimos el tope de la pila por el marco  $\mathbf{O}(v_1, \square)$ .

$$\overline{\mathbf{O}(\square, e_2); Pv_1 \rightarrow_h \mathbf{O}(v_1, \square); P \succ e_2}$$

Si se devuelve un valor a la pila que tiene como tope el marco  $\mathbf{O}(v_1, \square)$  entonces podemos devolver al resto de la pila el resultado de la suma de ambos valores.

$$\overline{\mathbf{O}(v_1, \square); P \prec v_2 \rightarrow_h P \prec v_1 \mathbf{o} v_2}$$

**Condicional** Para evaluar la expresión  $if(e_1, e_2, e_3)$  agregamos el marco  $if(\square, e_2, e_3)$  al tope de la pila y evaluamos  $e_1$ .

$$\overline{P \succ if(e_1, e_2, e_3) \rightarrow_h if(\square, e_2, e_3); P \succ e_1}$$

Si se regresa **True** a la pila con el marco  $if(\square, e_2, e_3)$  en el tope, entonces evaluamos  $e_2$  con el resto de la pila.

$$\overline{if(\square, e_2, e_3); P \prec \mathbf{True} \rightarrow_h P \succ e_2}$$

Si se regresa **False** a la pila con el marco  $if(\square, e_2, e_3)$  en el tope, entonces evaluamos  $e_3$  con el resto de la pila.

$$\overline{if(\square, e_2, e_3); P \prec \mathbf{False} \rightarrow_h P \succ e_3}$$

**Asignaciones locales** Si se quiere evaluar la expresión  $\text{let}(e_1, x.e_2)$  con la pila  $P$  entonces se evalúa  $e_2$  en donde se sustituyen las apariciones de  $x$  por  $e_1$  con la misma pila.

$$\frac{}{P \succ \text{let}(e_1, x.e_2) \rightarrow_h P \succ e_2[x := e_1]}$$

**Aplicación de función** Para evaluar una aplicación  $\text{app}(e_1, e_2)$  en una pila  $P$  se agrega el marco  $\text{app}(\square, e_2)$  como tope y se evalúa  $e_1$ .

$$\frac{}{P \succ \text{app}(e_1, e_2) \rightarrow_h \text{app}(\square, e_2); P \succ e_1}$$

Si se regresa un valor  $\text{fun}(x.e_1)$  a la pila con tope  $\text{app}(\square, e_2)$  entonces se quita el tope y se evalúa  $e_1$  sustituyendo  $x$  por  $e_2$ .

$$\frac{}{\text{app}(\square, e_2); P \prec \text{fun}(x.e_1) \rightarrow_h P \succ e_1[x := e_2]}$$

Si se regresa un valor  $\text{recfun}(f.x.e_1)$  a la pila con tope  $\text{app}(\square, e_2)$  entonces se quita el tope y se evalúa  $e_1$  sustituyendo  $f$  por su punto fijo y  $x$  por  $e_2$ .

$$\frac{}{\text{app}(\square, e_2); P \prec \text{recfun}(f.x.e_1) \rightarrow_h P \succ e_1[f := \text{fix}(f.x.e_1), x := e_2]}$$

**El operador de punto fijo** Para evaluar la expresión  $\text{fix}(f.e)$  en la pila  $P$  se evalúa  $e$  sustituyendo  $f$  por  $\text{fix}(f.e)$ .

$$\frac{}{P \succ \text{fix}(f.e) \rightarrow_h P \succ e[f := \text{fix}(f.e)]}$$

Con las definiciones discutidas anteriormente tenemos una primera implementación de un modelo de ejecución para *MinHs*. Este modelo nos es de gran utilidad para ilustrar paso a paso el estado de nuestro programa haciendo explícitas las operaciones pendientes de evaluar en nuestra pila y la expresión que se está evaluando en el paso.

La única entidad que podemos regresar a los marcos contenidos en la pila son los valores dado que no se pueden reducir a ningún otro estado, desahogando la pila y continuando con la evaluación de los marcos restantes.

A continuación presentamos dos ejemplos para ilustrar los marcos, estados y reglas de evaluación definidas anteriormente.

**Ejemplo 1.1** (Ejecución máquina  $\mathcal{J}$  con *closures*). Para ver el funcionamiento de la máquina  $\mathcal{J}$  vamos a evaluar la siguiente expresión:

```
let x = false in
  let y = 4 in
    if (x then y + 1 else y - 1)
  end
end
```

En sintaxis abstracta:

$$\text{let}(\mathbf{False}, x.\text{let}(4, y.\text{if}(x, \text{sum}(y, 1), \text{sub}(y, 1))))$$

La evaluamos en la máquina  $\mathcal{J}$ .

$$\begin{aligned}
& \diamond \succ \text{let}(\mathbf{False}, x.\text{let}(4, y.\text{if}(x, \text{sum}(y, 1), \text{sub}(y, 1)))) \\
& \diamond \succ \text{let}(4, y.\text{if}(x, \text{sum}(y, 1), \text{sub}(y, 1)))[x := \mathbf{False}] \\
& \diamond \succ \text{let}(4, y.\text{if}(x, \text{sum}(y, 1), \text{sub}(y, 1)))[x := \mathbf{False}] \\
& \diamond \succ \text{let}(4, y.\text{if}(\mathbf{False}, \text{sum}(y, 1), \text{sub}(y, 1))) \\
& \diamond \succ \text{let}(4, y.\text{if}(\mathbf{False}, \text{sum}(y, 1), \text{sub}(y, 1))) \\
& \diamond \succ \text{if}(\mathbf{False}, \text{sum}(y, 1), \text{sub}(y, 1))[y := 4] \\
& \diamond \succ \text{if}(\mathbf{False}, \text{sum}(4, 1), \text{sub}(4, 1)) \\
\\
& \text{if}(\square, \text{sum}(4, 1), \text{sub}(4, 1)) : \diamond \succ \mathbf{False} \\
& \text{if}(\square, \text{sum}(4, 1), \text{sub}(4, 1)) : \diamond \prec \mathbf{False} \\
& \quad \diamond \succ \text{sub}(4, 1) \\
& \quad \text{sub}(\square, 1) : \diamond \succ 4 \\
& \quad \text{sub}(\square, 1) : \diamond \prec 4 \\
& \quad \text{sub}(4, \square) : \diamond \succ 1 \\
& \quad \text{sub}(4, \square) : \diamond \prec 1 \\
& \quad \diamond \succ 4 - 1 \\
& \quad \diamond \prec 3
\end{aligned}$$

**Ejemplo 1.2** (Ejecución máquina  $\mathcal{J}$  con *closures*). Para ver el funcionamiento de la máquina  $\mathcal{J}$  vamos a evaluar la siguiente expresión:

```
let x = fun z -> z + 1 in
  let y = 4 in
    x y
  end
end
```

En sintaxis abstracta:

$$\text{let}(\text{lam}(z.\text{sum}(z, 1)), x.\text{let}(4, y.\text{app}(x, y)))$$

La evaluamos en la máquina  $\mathcal{J}$ .

$$\begin{aligned}
 & \diamond \succ \text{let}(\text{lam}(z.\text{sum}(z, 1)), x.\text{let}(4, y.\text{app}(x, y))) \\
 & \diamond \succ \text{let}(4, y.\text{app}(x, y))[x := \text{lam}(z.\text{sum}(z, 1))] \\
 & \diamond \succ \text{let}(4, y.\text{app}(\text{lam}(z.\text{sum}(z, 1)), y)) \\
 & \diamond \succ \text{app}(\text{lam}(z.\text{sum}(z, 1)), y)[y := 4] \\
 & \diamond \succ \text{app}(\text{lam}(z.\text{sum}(z, 1)), 4) \\
 \text{app}(\square, 4) : & \diamond \succ \text{lam}(z.\text{sum}(z, 1)) \\
 \text{app}(\square, 4) : & \diamond \prec \text{lam}(z.\text{sum}(z, 1)) \\
 & \diamond \succ \text{sum}(z, 1)[z := 4] \\
 & \diamond \succ \text{sum}(4, 1) \\
 \text{sum}(\square, 1) : & \diamond \succ 4 \\
 \text{sum}(\square, 1) : & \diamond \prec 4 \\
 \text{sum}(\_, \text{square}) : & \diamond \succ 1 \\
 \text{sum}(\_, \text{square}) : & \diamond \prec 1 \\
 & \diamond \succ 4 + 1 \\
 & \diamond \prec 5
 \end{aligned}$$

## 2 La Máquina $\mathcal{J}$

La máquina  $\mathcal{J}$  es una extensión de la máquina  $\mathcal{H}$  con la introducción de un contexto para las variables al que denominamos **entorno**, este entorno será el encargado de registrar los nombres de las variables así como sus valores para ser referidas en las operaciones de sustitución (**let**, **lam**, **fix**, **fun**, **app**).

Ésto con el objetivo de tener una implementación eficiente para la sustitución, operación que hasta el momento posee una complejidad lineal sobre el tamaño de la expresión que queremos evaluar.

Con pequeños ajustes podemos redifinir este nuevo modelo partiendo de las reglas definidas para  $\mathcal{H}$  los cuales discutiremos a continuación.

**Observación.** La implementación de la máquina  $\mathcal{J}$  cambiará el paradigma de evaluación por uno de tipo **ansioso** para simplificar los computos almacenados en los marcos.

## 2.1 Marcos

**Definición 2.1** (Marcos). En la máquina  $\mathcal{J}$  se usa el mismo conjunto de marcos que los presentados para la máquina  $\mathcal{H}$  con el siguiente cambio para los operadores **let** y **app** para el conjunto de marcos de la máquina  $\mathcal{J}$ :

**Asignaciones locales**

$$\frac{}{\text{let}(\square, x.e_2) \text{ marco}}$$

**Aplicación de función**

$$\frac{}{\text{app}(f, \square) \text{ marco}}$$

De esta forma la evaluación **ansiosa** obtendrá el valor a sustituir en la expresión para posteriormente evaluarla.

## 2.2 Ambientes

**Definición 2.2** (Ambientes). Un ambiente es una estructura que almacena asignaciones de variables con su valor y la definimos inductivamente como:

$$\frac{}{\bullet \text{ env}} \text{ vacio} \quad \frac{x \text{ var} \quad v \text{ valor} \quad \bullet \text{ env}}{x \leftarrow v ; \bullet \text{ env}} \text{ asig}$$

La forma de acceder a los elementos del ambiente es mediante el nombre de la variable, entonces si el ambiente  $\mathbb{E}$  tiene la asignación  $x \leftarrow v$ . Podemos acceder al valor de  $x$  como  $\mathbb{E}[x]$  y el resultado es  $v$ .

En caso de tener mas de una asignación sobre el mismo nombre de variable  $\mathbb{E}[x]$  nos regresa la primera aparición de  $x$  en el ambiente. Por ejemplo en el ambiente  $\mathbb{E} =_{def} x \leftarrow v_1; x \leftarrow v_2; \mathbb{E}$  la operación  $\mathbb{E}[x]$  nos arroja como resultado  $v_1$ .

## 2.3 Estados

**Definición 2.3** (Estados de la máquina  $\mathcal{J}$ ). Los estados ahora son una relación ternaria de una pila de control  $P$  un ambiente  $\bullet$  y una expresión  $e$ , denotados como:

- **Estados de evaluación:**  $P \mid \mathbb{E} \succ e$
- **Estados de retorno:**  $P \mid \mathbb{E} \prec e$

Un estado inicial es de la forma:

$$\diamond \mid \bullet \succ e$$

Mientras que los estados finales son de la forma:

$$\diamond \mid \mathbb{E} \prec v$$

Notemos que en los estados finales no importa que está guardado en el ambiente solo importa que la pila de control esté vacía.

## 2.4 Transiciones

**Definición 2.4** (Transiciones de variables en la máquina  $\mathcal{J}$ ). Se definen las transiciones de la máquina sin utilizar la operación de sustitución.

**Variables** En la máquina  $\mathcal{H}$  una variable representaba un estado bloqueado, pues la única forma de llegar a ella era que se tratara de una variable libre. En la máquina  $\mathcal{J}$  como no aplicamos sustitución tenemos que evaluar las variables buscándolas en el ambiente. Lo que definimos con la siguiente regla:

$$\frac{}{P \mid \mathbb{E} \succ x \rightarrow_j P \mid \mathbb{E} \prec \mathbb{E}[x]}$$

Nuestro contexto nos ayuda a registrar variables y sus valores asociados de tal forma que cuando ejecutamos una instrucción que necesita sustituir el valor de una variable en otra expresión. Dicha expresión define un alcance para la variable como podemos ver en el siguiente ejemplo:

$$\text{let}(e_1, x.e_2)$$

Se debe agregar al ambiente la asignación  $x \leftarrow e_1$  y evaluar  $e_2$  para obtener el resultado. Pero una vez que termine la evaluación de la aplicación de función esta asignación debe quitarse del ambiente pues ya terminó su alcance.

Para lograrlo es necesario extender la definición de la pila de ejecución para que guarde no sólo marcos sino también ambientes.

**Definición 2.5** (Pila de control para  $\mathcal{J}$ ).

$$\frac{}{P \text{ pila}} \text{ vacia} \quad \frac{\mathfrak{m} \text{ marco} \quad P \text{ pila}}{\mathfrak{m}; P \text{ pila}} \text{ top} \quad \frac{\mathbb{E} \text{ env} \quad P \text{ pila}}{\mathbb{E} ; P \text{ pila}} \text{ top}$$

De esta forma cuando evaluemos una expresión que genera un alcance distinto se agrega el ambiente anterior a la pila de control y se definen las nuevas asignaciones dentro

del ambiente actual. Una vez que termine la ejecución y se regrese un valor a la pila con un ambiente en el tope, continuamos con la ejecución del programa tomando ese ambiente como actual.

**Definición 2.6** (Transiciones con alcance en la máquina  $\mathcal{J}$ ). Ahora reescribimos los casos que involucran usar sustitución para su evaluación.

### Asignaciones locales

$$\frac{}{P \mid \mathbb{E} \succ let(e_1, x.e_2) \rightarrow_j let(\square, x.e_2); P \mid \mathbb{E} \succ e_1}$$

$$\frac{}{let(\square, x.e_2) ; P \mid \mathbb{E} \prec v \rightarrow_j \bullet ; P \mid x \leftarrow v ; \mathbb{E} \succ e_2}$$

### Aplicación de función

$$\frac{}{app(\square, e_2); P \mid \mathbb{E} \prec v_1 \rightarrow_j app(v_1, \square); P \mid \mathbb{E} \succ e_2}$$

$$\frac{}{app(fun(x.e_1), \square) ; P \mid \mathbb{E} \prec v_2 \rightarrow_j \mathbb{E} ; P \mid x \leftarrow v_2; \mathbb{E} \succ e_1}$$

$$\frac{}{app(recfun(f.x.e_1), \square); P \mid \mathbb{E} \prec v_2 \rightarrow_j \mathbb{E}; P \mid f \leftarrow fix(f.x.e_1); x \leftarrow v_2; \mathbb{E} \succ e_1}$$

**Liberación del ambiente** Esta regla se agrega para liberar el ambiente de la pila de control.

$$\frac{}{\mathbb{E}; P \mid \mathbb{E}_1 \prec v \rightarrow_j P \mid \mathbb{E} \prec v}$$

Observemos como la evaluación del valor a almacenar en las asignaciones locales se volvió un punto estricto pues los ambientes solo almacenan valores.

**Ejemplo 2.1.** Ejecución de la máquina  $\mathcal{J}$  Tomemos como ejemplo la siguiente expresión de *MinHs*

```
let f = fun x => x + y + z in
  let y = 4 in
    let z = 10 in
      f 1
    end
  end
end
```

La cual nos da la siguiente representación en **sintaxis abstracta**:

$$let(lam(y.sum(x, sum(y, z))), f.let(4, y.let(10, z.app(f, 1))))$$

La evaluación en la máquina  $\mathcal{J}$  es de la siguiente forma:

```

◊ | • > let(lam(y...))
let(□, f.let(...)) : ◊ | • > lam(y.sum(x, sum(y, z)))
let(□, f.let(...)) : ◊ | • < lam(y.sum(x, sum(y, z)))
◊ | f ← lam(x.sum(x, sum(y, z))) : • > let(4, y.let(10, z.app(f, 1)))
◊ | f ← lam(x.sum(x, sum(y, z))) : • > let(4, y.let(10, z.app(f, 1)))
let(□, y.let(10, z.app(f, 1))) : ◊ | f ← lam(x.sum(x, sum(y, z))) : • > 4
let(□, y.let(10, z.app(f, 1))) : ◊ | f ← lam(x.sum(x, sum(y, z))) : •E1 < 4
E1 : ◊ | y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • > let(10, z.app(f, 1))
let(□, z, app(f, 1)) : E1 : ◊ | y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • > 10
let(□, z, app(f, 1)) : E1 : ◊ | y ← 4 : f ← lam(x.sum(x, sum(y, z))) : •E2 < 10
E2 : E1 : ◊ | z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • > app(f, 1)
app(□, 1) : E2 : E1 : ◊ | z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • > f
app(□, 1) : E2 : E1 : ◊ | z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • > lam(x.sum(x, sum(y, z)))
app(□, 1) : E2 : E1 : ◊ | z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • < lam(x.sum(x, sum(y, z)))
app(lam(y.sum(x, sum(y, z))), □) : E2 : E1 : ◊ | z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • > 1
app(lam(y.sum(x, sum(y, z))), □) : E2 : E1 : ◊ | z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : •E3 < 1
E3 : E2 : E1 : ◊ | y ← 1 : z ← 10 : y ← 4 : f ← lam(y.sum(x, sum(y, z))) : • < sum(x, sum(y, z))
sum(□, sum(y, z)) : E3 : E2 : E1 : ◊ | x ← 1 : z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • < x
sum(□, sum(y, z)) : E3 : E2 : E1 : ◊ | x ← 1 : z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • < 1
sum(□, sum(y, z)) : E3 : E2 : E1 : ◊ | x ← 1 : z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • > 1
sum(1, □) : E3 : E2 : E1 : ◊ | x ← 1 : z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • < sum(y, z)
sum(□, z) : sum(1, □) : E3 : E2 : E1 : ◊ | x ← 1 : z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • < y
sum(□, z) : sum(1, □) : E3 : E2 : E1 : ◊ | x ← 1 : z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • < 4
sum(□, z) : sum(1, □) : E3 : E2 : E1 : ◊ | x ← 1 : z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • > 4
sum(4, □) : sum(1, □) : E3 : E2 : E1 : ◊ | x ← 1 : z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • > z
sum(4, □) : sum(1, □) : E3 : E2 : E1 : ◊ | x ← 1 : z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • < 10
sum(1, □) : E3 : E2 : E1 : ◊ | x ← 1 : z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • < 4 + 10
E3 : E2 : E1 : ◊ | x ← 1 : z ← 10 : y ← 4 : f ← lam(x.sum(x, sum(y, z))) : • < 14 + 1
E2 : E1 : ◊ | E : 3 : • < 14 + 1
E1 : ◊ | E2 : • < 15
◊ | E1 : • < 15

```

## 2.5 Alcance

Es importante detenernos a estudiar los diferentes alcances que la máquina  $\mathcal{H}$  y la máquina  $\mathcal{J}$  introducen con su definición. Tomemos como ejemplo la siguiente expresión de  $MinHs$

```

let x = 0 in
  let x = 1 in
    sum(x, 1)
  end
end

```

Si evaluamos la expresión utilizando la definición de la máquina  $\mathcal{H}$  el resultado será 1, dado que el alcance de esta máquina es estático, es decir, está limitado por la definición de los marcadores **in** y **let**. Esta implementación utiliza la operación de sustitución al evaluar un operador **let** por lo que obtendremos **sum(0,1) = 1**.

Si por el contrario evaluamos la expresión con la definición de la máquina  $\mathcal{J}$  el resultado será 2 dado que esta máquina implementa un alcance dinámico, es decir, el alcance de las variables que definimos para ser sustituidas es todo el programa, así cuando encontramos una aparición de dicha variable se extrae del contexto el primer valor que coincida con el nombre. en este caso en el contexto tendremos  $E = x \leftarrow 1 : x \leftarrow 0$  y la suma se evaluará como  $\text{sum}(1,1) = 2$ .

**Definición 2.7** (Alcance estático). En un lenguaje con alcance estático, el alcance de una variable es la región en la cual se encuentra definida.

**Definición 2.8** (Alcance dinámico). En un lenguaje con alcance dinámico, el alcance de un identificador es todo el programa, es decir, se toma la última asignación hecha al mismo.

## 2.6 Closures

La descripción de alcances discutida anteriormente nos muestra un problema en la implementación de nuestras máquinas dado que los resultados obtenidos al evaluar la misma expresión son distintos. Esto se debe de corregir para la definición de la máquina  $\mathcal{J}$  mediante la definición de **Closures** que encapsulan el contexto de evaluación y la expresión para la que dicho contexto será aplicado cuando se evalúe.

Los **closures** nos ayudan a imitar la definición de un alcance estático en la implementación de  $\mathcal{J}$ , para ésto deberemos modificar las reglas de evaluación.

**Definición 2.9** (Closure). Un *Closure* es una pareja de una expresión de función de y un ambiente que se denota como:

$$\ll \mathbb{E}, f \gg$$

y se interpreta como, que el ambiente adecuado para evaluar la función  $f$  es  $\mathbb{E}$ , de esta forma se respeta el ambiente en el que se define una función para usar el mismo en su ejecución y de esta forma definir una evaluación con alcance estático.

Ahora se modifican las transiciones definidas en la sección anterior para que usen *closures* y modelen una evaluación con alcance estático, en lugar de la evaluación con alcance dinámico presentada anteriormente.

**Definición 2.10** (Transición para funciones). En lugar de regresar las funciones como una expresión, se guardará como una pareja de la expresión y el ambiente en el que fue definido.

$$\begin{array}{c}
\frac{}{P \mid \mathbb{E} \succ \text{fun}(x.e) \rightarrow_j P \mid \mathbb{E} \prec \ll \mathbb{E}, x.e \gg} \\[10pt]
\frac{}{app(\square, e_2); P \mid \mathbb{E} \prec \ll \mathbb{E}_f, x.e_1 \gg \rightarrow_j app(\ll \mathbb{E}_f, x.e_1 \gg, \square); P \mid \mathbb{E} \succ e_2} \\[10pt]
\frac{}{app(\ll \mathbb{E}_f, x.e_1 \gg, \square); P \mid \mathbb{E} \prec v \rightarrow_j \mathbb{E}; P \mid x \leftarrow v; \mathbb{E}_f \succ e_1} \\[10pt]
\frac{}{P \mid \mathbb{E} \succ \text{recfun}(f.x.e) \leftarrow_j P \mid \mathbb{E} \prec fix(f. \ll \mathbb{E}, x.e \gg)} \\[10pt]
\frac{}{app(\square, e_2); P \mid \mathbb{E} \prec fix(f. \ll_f, x.e_1 \gg) \rightarrow_j app(f. \ll_f, x.e_1 \gg), \square); P \mid \mathbb{E} \succ e_2} \\[10pt]
\frac{}{app(fix(f. \ll_f, x.e_1 \gg), \square); P \mid \mathbb{E} \prec v \rightarrow_j \mathbb{E}; P \mid x \leftarrow v; f \leftarrow (f.f, x.e_1); f \succ e_1}
\end{array}$$

Con estas reglas se generalizan los *closures* en la evaluación de una función para que de esta forma el ambiente con el que se ejecutan en la aplicación sea el mismo ambiente en el que se definió la función y las variables tomen el valor esperado según el alcance estático.

### Ejemplo 2.2.

```

let y = 4 in
  let z = 10 in
    let f = fun x => x + y + z in
      f 1
    end
  end
end

```

La cual nos da la siguiente representación en **sintaxis abstracta**:

```
let(4, y.let(10, z.let(lam(x.sum(x, sum(y, z)), f.app(f, 1)))))
```

La evaluación en la máquina  $\mathcal{J}$  es de la siguiente forma:

$  \begin{aligned}  & \diamond   \bullet \\  & let(\square, y, let...) : \diamond   \bullet \\  & let(\square, y, let...) : \diamond   \bullet \\  & \quad \bullet : \diamond   y \leftarrow 4 : \bullet \\  & let(\square, z, let...) : \bullet : \diamond   y \leftarrow 4 : \bullet \\  & let(\square, z, let...) : \bullet : \diamond   \underline{y \leftarrow 4 : \bullet} E_1 \\  & \quad E_1 : \bullet : \diamond   z \leftarrow 10 : E_1 \\  & let(\square, f, app...) : E_1 : \bullet : \diamond   z \leftarrow 10 : E_1 \\  & let(\square, f, app...) : E_1 : \bullet : \diamond   z \leftarrow 10 : E_1 E_2 \\  & E_2 : E_1 : \bullet : \diamond   f \leftarrow \ll E_2, lam(x.sum(x, sum(y, z))) \gg : E_2 \\  & app(\square, 1) : E_2 : E_1 : \bullet : \diamond   f \leftarrow \ll E_2, lam(x.sum(x, sum(y, z))) \gg : E_2 \\  & app(\square, 1) : E_2 : E_1 : \bullet : \diamond   f \leftarrow \ll E_2, lam(x.sum(x, sum(y, z))) \gg : E_2 \\  & app(\ll E_2, lam(y.sum(x, sum(y, z))) \gg, \square) : E_2 : E_1 : \bullet : \diamond   f \leftarrow \ll E_2, lam(x.sum(x, sum(y, z))) \gg : E_2 \\  & app(\ll E_2, lam(y.sum(x, sum(y, z))) \gg, \square) : E_2 : E_1 : \bullet : \diamond   \underline{f \leftarrow \ll E_2, lam(x.sum(x, sum(y, z))) \gg : E_2} E_3 \\  & \quad E_3 : E_2 : E_1 : \bullet : \diamond   x \leftarrow 1 : E_2 \\  & \quad sum(\square, sum(y, z)) : E_3 : E_2 : E_1 : \bullet : \diamond   x \leftarrow 1 : E_2 \\  & \quad sum(\square, sum(y, z)) : E_3 : E_2 : E_1 : \bullet : \diamond   x \leftarrow 1 : E_2 \\  & \quad sum(\square, sum(y, z)) : E_3 : E_2 : E_1 : \bullet : \diamond   x \leftarrow 1 : E_2 \\  & \quad sum(1, \square) : E_3 : E_2 : E_1 : \bullet : \diamond   x \leftarrow 1 : E_2 \\  & \quad sum(\square, z) : sum(1, \square) : E_3 : E_2 : E_1 : \bullet : \diamond   x \leftarrow 1 : E_2 \\  & \quad sum(\square, z) : sum(1, \square) : E_3 : E_2 : E_1 : \bullet : \diamond   x \leftarrow 1 : E_2 \\  & \quad sum(4, \square) : sum(1, \square) : E_3 : E_2 : E_1 : \bullet : \diamond   x \leftarrow 1 : E_2 \\  & \quad sum(4, \square) : sum(1, \square) : E_3 : E_2 : E_1 : \bullet : \diamond   x \leftarrow 1 : E_2 \\  & \quad sum(1, \square) : E_3 : E_2 : E_1 : \bullet : \diamond   : x \leftarrow 1 : E_2 \\  & \quad E_3 : E_2 : E_1 : \bullet : \diamond   : x \leftarrow 1 : E_2 \\  & \quad E_2 : E_1 : \bullet : \diamond   E_3 \\  & \quad E_1 : \bullet : \diamond   E_2 \\  & \quad \bullet : \diamond   E_1 \\  & \quad \diamond   \bullet  \end{aligned}  $	$\succ$ $let(4, y, let(10, z, let(lam(x.sum(x, sum(y, z)), f.app(f, 1))))$ $4$ $4$ $let(10, z, let(lam(x.sum(x, sum(y, z)), f.app(f, 1))))$ $10$ $10$ $let(lam(x.sum(x, sum(y, z)), f.app(f, 1)))$ $lam(x.sum(x, sum(y, z)))$ $\ll E_2, lam(x.sum(x, sum(y, z))) \gg$ $app(f, 1)$ $f$ $\ll E_2, lam(x.sum(x, sum(y, z))) \gg$ $\ll E_2, lam(x.sum(x, sum(y, z))) \gg$ $1$ $1$ $sum(x, sum(y, z))$ $x$ $x$ $1$ $sum(y, z)$ $y$ $4$ $z$ $10 + 4$ $14 + 1$ $15$ $15$ $15$
---	--

Como podemos observar el resultado obtenido ahora es el mismo en ambas máquinas, evaluando a 15.

De esta forma concluimos el estudio de la implementación de las máquinas abstractas para *MinHs*. A continuación se encuentran los ejercicios de comprensión para el lector.

### 3 Ejercicios para el lector

**Ejemplo 3.1.** Dada la siguiente expresión de *MinHs* contesta lo siguiente:

```
let x = False in
    let y = fun z => leq(z,0) in
        let x = True in
            if x then y 0 else False
        end
    end
end
```

- Evalúa la expresión de acuerdo a la definición de la máquina  $\mathcal{H}$ .
- Evalúa la expresión de acuerdo a la definición de la máquina  $\mathcal{J}$ .
- Los resultados obtenidos son los mismos o son diferentes?.
- Evalúa la expresión utilizando la definición extendida para la máquina  $\mathcal{J}$  con **Closures**.
- El resultado obtenido es el mismo?.

**Ejemplo 3.2.** Dada la siguiente expresión de *MinHs* contesta lo siguiente:

```
let recfun pow x =
    if x = 0 then 1 else x * pow x - 1 in
        pow 3
    end
```

- Evalúa la expresión de acuerdo a la definición de la máquina  $\mathcal{H}$ .
- Evalúa la expresión de acuerdo a la definición de la máquina  $\mathcal{J}$ .
- Los resultados obtenidos son los mismos o son diferentes?.
- Evalúa la expresión utilizando la definición extendida para la máquina  $\mathcal{J}$  con **Closures**.
- El resultado obtenido es el mismo?.

**Ejemplo 3.3.** Dada la siguiente expresión de *MinHs* contesta lo siguiente:

```
let x = True in
    let x = False in
        if x then 1 else 0
    end
end
```

- Evalúa la expresión de acuerdo a la definición de la máquina  $\mathcal{H}$ .
- Evalúa la expresión de acuerdo a la definición de la máquina  $\mathcal{J}$ .
- Los resultados obtenidos son los mismos o son diferentes?.
- Evalúa la expresión utilizando la definición extendida para la máquina  $\mathcal{J}$  con **Closures**.
- El resultado obtenido es el mismo?.

## Capítulo 9

### TinyC



Figura 9.1: Denis Ritchie, creador del lenguaje de programación C (nótese que me estaba quedando sin ideas y decidí hacerlo un gato para llenar la imagen del capítulo).

Hasta ahora nuestro caso de estudio con el lenguaje nos fue de utilidad para ilustrar el paradigma funcional. La realidad es que muchos de los lenguajes que utilizamos actualmente no se implementan de esta forma.

Es aquí cuando haremos un cambio abrupto en nuestro caso de estudio para poner nuestra atención en el paradigma **Procedimental** cuyo primer ejemplo sera una implementación de un lenguaje de programación basado en *C* conocido como **TinyC**<sup>1</sup>.

---

<sup>1</sup>Las definiciones que visitaremos en este capítulo fueron extraidas de: Enríquez Mendoza J., Lenguajes

Este tipo de lenguajes suponen el manejo de **estados** los cuales son afectados por el contexto del programa, los cambios en la memoria y las instrucciones que se ejecutan en un determinado momento, Similar a un autómata o a una máquina de estados, lo cual supone una ligera desventaja al momento de querer demostrar propiedades sobre éste.

## Objetivo

En este capítulo comenzamos el estudio del paradigma **Procedimental**, para ésto nuestra interés principal será el de definir el primer caso de estudio conocido como **TinyC**.

## Planteamiento

Se revisarán los componentes básicos de este lenguaje, a saber: **Sintáxis, Semántica Operacional** mediante la implementación de la **Máquina C** incluyendo sus **Marcos, Estados Transiciones**. Adicionalmente se discutirá la **Semántica Estática** de **TinyC**.

# 1 Sintáxis

**Definición 1.1** (Sintaxis Concreta de ).

```
progam ::= global-decs stmt
global-decs ::= ε | global-dec global-decs
global-dec ::= fun-dec | var-dec
var-decs ::= ε | var-dec var-decs
var-dec ::= type ident = expr;
fun-dec ::= type ident (arguments) stmt;
stmt ::= expr; | if expr then stmt else stmt; | if expr then stmt;
        | expr; | {var-decs stmts} | (expr) stmt
stmts ::= ε | stmt stmts
expr ::= num | ident | asig | expr + expr | expr - expr
        | expr > expr | expr < expr | ident(exprs)
asig ::= ident = expr
exprs ::= expr | expr, exprs
arguments ::= ε | type ident, arguments
type ::= Int | Bool
```

De la definición anterior es importante notar la omisión entera de apuntadores dado que el tema escapa del enfoque de este capítulo quedándonos únicamente con un subconjunto de expresiones del lenguaje de programación **C**.

En **TinyC** tendremos las **declaraciones** de variables y funciones que se pueden identificar por nombre, las **expresiones** aritméticas y lógicas para construir la lógica de los programas y por último las **sentencias** encargadas de la ejecución y control del estado del programa.

También es importante remarcar la diferencia que la introducción de las variables en este paradigma suponen para el estado de un programa, dado que el valor que contienen puede ser modificando alterando así el estado mismo del programa en cualquier punto del mismo cosa que no ocurría con la introducción de las variables en la sintaxis de orden superior para **MinHs**.

**Ejemplo 1.1.** Escribe un programa en **TinyC** que implementa una función que nos permita calcular el **n-ésimo** número de la sucesión de Fibonacci.

```
fibonacci(Int n){  
  
    Int i = ;  
    Int pre = 1;  
    Int post = 1;  
    Int fib;  
  
    while(i < n){  
        fib = aux2 + aux1;  
        pre = post;  
        post = fib;  
    }  
    return fib;  
};  
  
Int result = fibonacci(9);
```

**Observación.** Observese que omitimos la definición entera de la **sintaxis abstracta**. El lector debe interpretar que cada programa escrito en **TinyC** tiene un mapeo a una representación con un árbol de sintaxis abstracta que representaremos con la introducción de **marcos**.

## 2 Semántica operacional

### 2.1 La Máquina C

Para la semántica operacional de **TinyC** vamos a definir una máquina abstracta siguiendo la misma línea del capítulo anterior, donde contaremos con una pila para guardar los marcos de ejecución y cómputos pendientes y un contexto para las variables y los valores asignados a éstas.

### 2.2 Marcos

Para la sección de expresiones vamos a tener un mapeo directo con los marcos para los operadores de la máquina  $\mathcal{J}$  dado que la categoría *expr* de **TinyC** también regresa un valor al ser evaluadas.

Una diferencia importante a notar es la manera en la que las funciones son declaradas en **MinHs**. Aquí la declaración se hace mediante funciones anónimas que pueden ir en cualquier parte del programa, mientras que en **TinyC** y en **C** deben ser declaradas antes de ser llamadas. Adicionalmente estas declaraciones de funciones pueden ser multiparamétricas mientras que en **TinyC** solo pueden declararse un valor a la vez.

**Definición 2.1** (Marcos para la pila de control de la máquina  $\mathcal{C}$ ). Se definen los siguientes marcos que usaremos en la pila de control de la máquina  $\mathcal{C}$

#### Declaraciones

$$\frac{}{vardec(T, x, \square) \text{ marco}}$$

#### Asignaciones

$$\frac{}{asig(x, \square) \text{ marco}}$$

#### Secuencia

$$\frac{}{secu(\square, e_2) \text{ marco}}$$

#### Condicionales

$$\frac{}{if(\square, e_2, e_3) \text{ marco}}$$

$$\frac{}{if(\square, e_2) \text{ marco}}$$

#### Return

$$\frac{}{return(\square) \text{ marco}}$$

#### Llamada a función

$$\frac{}{call(f, \square, e_2, \dots, e_n) \text{ marco}} \quad \dots \quad \frac{}{call(f, v_1, v_2, \dots, \square) \text{ marco}}$$

## 2.3 Estados

Esta Categoría supone una diferencia sustancial con las máquinas abstractas hasta ahora estudiadas. Tendremos una categoría para la pila de compútos pendientes y dos categorías para el contexto de evaluación de las variables, el primero local representando el alcance de las variables declaradas en el cuerpo de una función y uno global para aquellas declaraciones cuyo contexto sea todo el programa.

**Definición 2.2** (Estados de la máquina  $\mathcal{C}$ ). Los estados de la máquina  $\mathcal{C}$  son de la siguiente forma:

$$P \mid L \mid G \succ e \quad P \mid L \mid G \prec e$$

En donde  $P$  es una pila de control,  $L$  y  $G$  son ambientes de variables y  $e$  es una expresión del lenguaje.

**Definición 2.3** (Consulta al ambiente). Definido recursivamente como sigue:

$$\frac{}{\bullet[x] = fail} \quad \frac{}{x \leftarrow v; \mathbb{E}[x] = v} \quad \frac{}{y \leftarrow v; \mathbb{E}[x] = \mathbb{E}[x]}$$

**Definición 2.4** (Modificación al ambiente). Definido recursivamente como sigue:

$$\frac{}{\bullet[x \rightarrow v] = fail} \quad \frac{}{x \leftarrow u; \mathbb{E}[x \rightarrow v] = x \leftarrow v \mathbb{E}} \\ \frac{}{y \leftarrow u; \mathbb{E}[x \rightarrow v] = \mathbb{E}[x \rightarrow v]}$$

## 2.4 Transiciones

**Definición 2.5** (Transiciones de la máquina  $\mathcal{C}$ ). Las transiciones de la máquina  $\mathcal{C}$  se definen en términos de los programas que se están evaluando. Como vimos en secciones anteriores en el caso de un programa no tiene un resultado final, es decir no se reduce a un valor. Por lo que se agrega un programa específico que indica el final de la ejecución, este programa es el programa vacío y se denota como:

$\perp$

y define el final del proceso de evaluación de una sentencia, por lo que los estados finales de la máquina  $\mathcal{C}$  son los que tienen la forma siguiente:

$$\diamond \mid L \mid G \prec \perp$$

Con lo que se definen las transiciones con las siguientes reglas:

## Declaraciones

$$\frac{}{P \mid L \mid G \succ vardec(T, x, e) \rightarrow_C vardec(T, x, \square); P \mid L \mid G \succ e}$$

$$\frac{G[x] = fail}{vardec(x, \square); P \mid L \mid G \prec v \rightarrow_C P \mid L \mid x \leftarrow v; G \prec \perp}$$

$$\frac{}{P \mid L \mid G \succ fundec(T, f, x_1 : T_1. \dots . x_n : T_n. e) \rightarrow_C P \mid L \mid f \leftarrow x_1. \dots . x_n. e; G \prec \perp}$$

## Asignación

$$\frac{}{P \mid L \mid G \succ asig(x, e) \rightarrow_C asig(x, \square); P \mid L \mid G \succ e}$$

$$\frac{G[x \mapsto v] = G'}{asig(x, \square); P \mid L \mid G \prec v \rightarrow_C P \mid L \mid G' \prec \perp}$$

$$\frac{G[x \mapsto v] = fail \quad L[x \mapsto v] = L'}{asig(x, \square); P \mid L \mid G \prec v \rightarrow_C P \mid L' \mid G \prec \perp}$$

## Variables

$$\frac{G[x] = v}{P \mid L \mid G \succ x \rightarrow_C P \mid L \mid G \prec v}$$

$$\frac{G[x] = fail \quad L[x] = v}{P \mid L \mid G \succ x \rightarrow_C P \mid L \mid G \prec v}$$

## Secuencia

$$\frac{}{P \mid L \mid G \succ secu(e_1, e_2) \rightarrow_C secu(\square, e_2); P \mid L \mid G \succ e_1}$$

$$\frac{}{secu(\square, e_2); P \mid L \mid G \prec \perp \rightarrow_C P \mid L \mid G \succ e_2}$$

## Condicionales

$$\frac{}{P \mid L \mid G \succ if(e_1, e_2, e_3) \rightarrow_C if(\square, e_2, e_3); P \mid L \mid G \succ e_1}$$

$$\frac{}{if(\square, e_2, e_3); P \mid L \mid G \prec true \rightarrow_C P \mid L \mid G \succ e_2}$$

$$\frac{}{if(\square, e_2, e_3); P \mid L \mid G \prec false \rightarrow_C P \mid L \mid G \succ e_3}$$

$$\frac{}{P \mid L \mid G \succ if(e_1, e_2) \rightarrow_C if(\square, e_2); P \mid L \mid G \succ e_1}$$

$$\frac{}{if(\square, e_2); P \mid L \mid G \prec true \rightarrow_C P \mid L \mid G \succ e_2}$$

$$\frac{}{if(\square, e_2); P \mid L \mid G \prec false \rightarrow_C P \mid L \mid G \prec \perp}$$

## While

$$\frac{}{P \mid L \mid G \succ while(e_1, e_2) \rightarrow_C P \mid L \mid G \succ if(e_1, secu(e_2, while(e_1, e_2)))}$$

Esta regla traduce la evaluación de un *while* a un *if* con un solo caso. Modela la siguiente regla equivalencia entre programas:

$$while(e_1)\{e_2\} \equiv if(e_1) \{ e_2 ; while(e_1)\{e_2\}\}$$

Obsérvese como la expresión *while* sigue apareciendo en el lado derecho. De esta forma se desdobra el ciclo tantas veces como se cumpla la condición.

## Llamada a función

$$\frac{}{P \mid L \mid G \succ call(f, e_1, \dots, e_n) \rightarrow_C call(\square, e_1, \dots, e_n); P \mid L \mid G \succ f}$$

$$\frac{}{call(\square, e_1, \dots, e_n); P \mid L \mid G \prec v \rightarrow_C call(v, \square, e_2, \dots, e_n); P \mid L \mid G \succ e_1}$$

⋮

$$\frac{}{call(x_1 \dots x_n.e, v_1, \dots, \square); P \mid L \mid G \prec v_n \rightarrow_C P \mid G \star L \mid x_1 \leftarrow v_1; \dots; x_n \leftarrow v_n; E \succ e}$$

Para la evaluación de una llamada a función, primero es necesario evaluar cada uno de los parámetros con los que se llama. Una vez que todos son valores, entonces se ejecuta el cuerpo de la función usando un ambiente vacío como ambiente principal y agregando a él los parámetros de la llamada. Y guardamos el ambiente principal anterior con un símbolo especial ( $\star$ ) que sirve como separado para saber en donde termina uno y comienza el otro.

**Return** El constructor `return` indica el final de la ejecución de una llamada a función.

$$\frac{}{P \mid L \mid G \succ return(\square) \rightarrow_C (\square); P \mid L \mid G \succ e}$$

$$\frac{}{return(\square); P \mid G \star L_1 \mid L_2 \prec v \rightarrow_C P \mid L_1 \mid G \prec v}$$

Cuando termina la ejecución de una llamada a función. Se restaura el ambiente global y nos deshacemos del local pues solo era necesario dentro del cuerpo de la función.

**Ejemplo 2.1** (Ejecución de la máquina C). Dado el siguiente programa de **TinyC** contesta lo siguiente:

```

fibonacci(Int n){

    Int i = ;
    Int pre = 1;
    Int post = 1;
    Int fib = 0;

    while(i < n){
        fib = pre + post;
        pre = post;
        post = fib;
        i++;
    }
    return fib;
};

Int result = fibonacci(3);

```

A) Evalúa el programa para obtener el resultado de acuerdo a las reglas de transición y estados definidos en la sección anterior.

Para facilitar la representación y la evaluación de la máquina C vamos a seccionar el programa para tener fragmentos de este, definimos entonces:

$$A = secu(vardec(Int, i, 1), secu(vardec(Int, pre, 1), secu(vardec(Int, post, 1), vardec(Int, fib, 0))))$$

$$B = secu(asig(fib, sum(pre, post)), secu(asig(pre, post), secu(asig(i, sum(i+1), asig(post, fib)))))$$

$$W = while(lt(i, n), B))$$

$$F = fundec(Int, fibonacci, n : Int.secu(A, secu(W, return(fib))))$$

$$P = secu(F, vardec(Int, result, call(fibonacci, 3)))$$

Se procede a evaluar el programa con la pila y ambos contextos de variables vacíos.

```

sum(□, post) : assig(fib, □) : secu(□, secu(asig(pre, post), ...)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L5 × 1
sum(1, post) : assig(fib, □) : secu(□, secu(asig(pre, post), ...)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L5 × post
sum(1, post) : assig(fib, □) : secu(□, secu(asig(pre, post), ...)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L5 × 2
assig(fib, □) : secu(□, secu(asig(pre, post), ...)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L5 × 3
secu(□, secu(asig(pre, post), ...)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | fib ← 3 : post ← 2 : pre ← 1 : i ← 2 : n ← 3 : •L6 × ⊥
secu(□, secu(asig(pre, post), ...)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L6 × ⊥
secu(□, secu(asig(i, sum(i + 1), ...)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L6 × secu(asig(pre, post), secu(asig(i, sum(i + 1), ...)))
asig(pre, □) : secu(□, secu(asig(i, sum(i + 1), ...)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L6 × post
asig(pre, □) : secu(□, secu(asig(i, sum(i + 1), ...)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L6 × 2
secu(□, secu(asig(i, sum(i + 1), ...)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | fib ← 3 : post ← 2 : pre ← 2 : i ← 2 : n ← 3 : •L7 × ⊥
secu(□, secu(asig(i, sum(i + 1), ...)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L7 × secu(asig(i, sum(i, 1), asig(post, fib)))
secu(□, asig(post, fib)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L7 × asig(i, sum(i, 1))
asig(i, □) : secu(□, asig(post, fib)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L7 × sum(i, 1)
asig(i, □) : secu(□, asig(post, fib)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L7 × sum(i, 1)
asig(i, □) : secu(□, asig(post, fib)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L7 × sum(i, 1)
sum(□, 1) : asig(i, □) : secu(□, asig(post, fib)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L7 × 2
sum(i, 1) : asig(i, □) : secu(□, asig(post, fib)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L7 × 2
sum(2, □) : asig(i, □) : secu(□, asig(post, fib)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L7 × 1
asig(i, □) : secu(□, asig(post, fib)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L7 × 3
secu(□, asig(post, fib)) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | fib ← 3 : post ← 2 : pre ← 2 : i ← 2 : n ← 3 : •L8 × ⊤
secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | fib ← 3 : post ← 2 : pre ← 2 : i ← 2 : n ← 3 : •L8 × ⊥
secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L8 × asig(post, fib)
asig(post, □) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L8 × fib
asig(post, □) : secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L8 × 3
secu(□, while(...)) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | fib ← 3 : post ← 2 : pre ← 2 : i ← 2 : n ← 3 : •L9 × ⊥
secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L9 × while(lt(i, n), B)
secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L9 × if(lt(i, n), secu(B, while(...)))
lt(□, n) : if(□, secu(B, while(...))) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L9 × i
lt(□, n) : if(□, secu(B, while(...))) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L9 × 3
lt(i, □) : if(□, secu(B, while(...))) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L9 × n
lt(i, □) : if(□, secu(B, while(...))) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L9 × 3
if(□, secu(B, while(...))) : secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L9 × false
secu(□, return(fib)) : vardec(Int, result, □) : o | L1★• | L9 × ⊥
vardec(Int, result, □) : o | L1★• | L9 × return(fib)
return(□) : vardec(Int, result, □) : o | • | 1 × fib
return(□) : vardec(Int, result, □) : o | • | L1 × 3
vardec(Int, result, □) : o | • | L1 × 3
o | • | result ← 3 : L1 × ⊥

```

Obsérvese que no regresamos el resultado al final de la evaluación si no que lo almacenamos en una de las variables que queda impresa.<sup>2</sup> en el contexto final.

Nuestra atención está no en el resultado que se obtiene en el último paso ( $\perp$ ) mas bien el estado de la memoria.

También es importante fijar nuestra atención en la notación que utilizamos durante toda la evaluación siendo de vital importancia el renombrado de los contextos para acortar la pila de variables.

Se pueden omitir pasos cuyas evaluaciones nos sean familiares<sup>2</sup>. En este caso la ejecución fue lo mas explícita posible para ilustrar el proceso pero el lector puede convencirse de que es tedioso y pesado por lo que permitiremos la omisión de dichas.

---

<sup>2</sup>Como las que estudiamos en el capítulo: **Máquinas Abstractas** correspondientes a las operaciones aritméticas/lógicas y expresiones de **TinyC**.

### 3 Ejercicios para el lector

**Ejercicio 3.1.** Utilizando la sintáxis definida para **TinyC** responde siguiente:

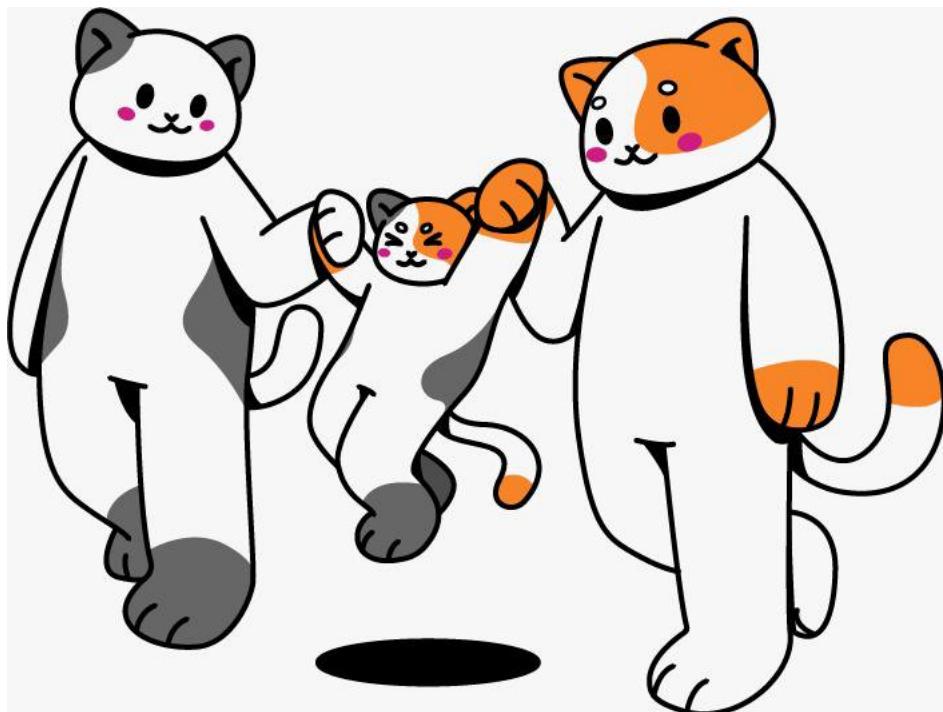
- Da la definición de un programa que dados dos enteros n y m revise que n es múltiplo de m.
- Da la definición de un programa que dado un número n revise su paridad.
- Da la definición de un programa que dados dos números n y m nos regrese el resultado de  $n^m$ .

**Ejercicio 3.2.** Utilizando la definición de la máquina C evalúa las expresiones del inciso anterior con los siguientes valores:

- 3 y 9
- 4
- 3 y 2

## Capítulo 10

# Herencia y subtipos



En este capítulo vamos a revisar la última categoría del enfoque de este manual: **La Orientación a Objetos**.

Esta subcategoría de la programación procedimental supone una ventaja al momento de reutilizar código que ya hemos definido anteriormente y que durante los últimos años supuso una revolución en la forma en la que se escriben y desarrollan los programas que corren en nuestros teléfonos y computadoras personales.

Lenguajes como **Java**, **Python**, **Ruby**, **Golang**, **Swift**, **C#** entre muchos otros implementan este mecanismo de "herencia", donde métodos, constructores y variables de clase de la llamada "clase padre" puede ser reutilizada por otra clase "clase hija".

Más aún, en este tipo de lenguajes la flexibilidad no solo se extiende a hacer uso de código previamente definido y heredado si no que se puede extender y sobrecargar sobre lo ya existente, proporcionando una herramienta poderosa al momento de desarrollar software.

## Objetivo

El objetivo de este capítulo es proporcionar una definición de la **La Orientación a Objetos** para el paradigma procedural mediante los mecanismos de subtipado para las diferentes categorías planteadas en la sintaxis de **TinyC**

## Planteamiento

En este capítulo se presentan las definiciones de **Orientación a Objetos** y **Subtipado**, enunciando las principales características de la primera y la aplicación de este concepto para cada nivel de la sintaxis de los programas del paradigma procedural en el caso de la segunda, finalmente se concluye el capítulo introduciendo el concepto de **Casting**<sup>1</sup>.

# 1 Orientación a objetos

Continuando con nuestro estudio sobre el paradigma procedural comenzaremos este capítulo abordando la **Orientación a Objetos** característica deseable para los programas clasificados como procedimentales y que supone una ventaja para la reutilización de código,

**La Orientación a Objetos** supone un consideración especial al momento de diseñar e implementar programas junto con el sistema de tipos para éste ya que la filosofía de desarrollo es diferente a lo que hemos estudiado hasta este momento.

Una de las piedras angulares de este paradigma es la introducción del concepto conocido como **Clase**.

**Definición 1.1** (Clase). Una **Clase** es el conjunto que conforma la plantilla o firma de un **Objeto**, contiene los atributos componentes que representan a dicho objeto como sus variables de clase y métodos que conforman la descripción y comportamiento del mismo.

**Definición 1.2** (Objeto). Un **objeto** es una instancia de una clase.

Las características que se comparten entre los lenguajes que implementan este paradigma incluyen:

<sup>1</sup>Las definiciones que revisaremos en este capítulo fueron extraídas de: Enríquez Mendoza J., Lenguajes de Programación Notas de Clase: Subtipado. Universidad Nacional Autónoma de México. 2022.

- **Herencia** Esta característica es de vital importancia permitiendo la definición única de una clase dejando que esta sea extendida o sobrecargada por las clases que heredan de ella, constituyendo un mecanismo para reutilizar código previamente escrito y adaptarlo al contexto particular donde se desean instanciar los objetos.
- **Encapsulamiento de datos** Esta característica comprende la seguridad de la información contenida en las variables y métodos de una clase, permitiendo que solo el objeto mismo sea el encargado de alterar la información contenida en él. Proporcionando seguridad y consistencia.
- **Polimorfismo** Esta característica permite que un objeto que ha heredado de una clase pueda ser identificado por cualquiera de los tipos que posea la cadena de herencia, pudiendo este ser, su mismo tipo o cualquiera de los tipos de la clase o clases padres.
- **Representaciones múltiples** Esta característica permite que dos objetos de una misma clase puedan presentar comportamiento distinto pues éste puede cambiar según el estado que posea cada uno.
- **Recursión abierta** Un objeto es capaz de invocarse a si mismo dentro de la definición de los métodos de su propia clase, utilizando las palabras reservadas como **self** ó **this**.

## 2 Subtipos

Los sistemas de tipos que hemos estudiado hasta este momento son restrictivos en el sentido de que muchas operaciones que parecerían intuitivas al desarrollar un programa no serían aceptadas por dicho sistema.

Particularmente operaciones como la suma (+) entre los tipos **Int** y **Float** por ejemplo.

Necesitamos relajar este sistema para tener mas flexibilidad sobre este tipo de casos, por esto se incluye la relación  $A \leq B$  que simboliza: ".<sup>A</sup> es subtipo de B".

**Definición 2.1** (Relación de subtipado). La relación de subtipado nos permite intercambiar a discreción un tipo A donde se espera un parámetro de tipo B  
Esta relación tiene las siguientes propiedades:

**Reflexividad**

$$\overline{T \leq T}$$

**Transitividad**

$$\frac{R \leq S \quad S \leq T}{R \leq T}$$

**Subsunción**

$$\frac{\Gamma \vdash e : S \quad S \leq T}{\Gamma \vdash e : T}$$

Esta propiedad expresa que si una expresión tiene tipo S y S es subtipo de T entonces puede usarse en cualquier contexto en donde sea necesaria una expresión de tipo T.

**Ejemplo 2.1** (Suma con subtipado). Para este ejemplo consideremos la expresión:

$$5,5 + 2$$

Donde podemos observar que  $5,5 : \text{Float}$  y  $2 : \text{Int}$ , suponemos entonces que los tipos **Float** e **Int** están relacionados bajo la relación de subtipado como

$$\text{Int} \leq \text{Float}$$

por lo que la derivación de tipos de la expresión anterior queda como sigue:

$$\frac{\frac{\frac{\emptyset \vdash 5,5 : \text{Float}}{\emptyset \vdash 3 : \text{Int}} \quad \text{Int} \leq \text{Float}}{\emptyset \vdash 3 : \text{Float}}}{\emptyset \vdash 5,5 + 3 : \text{Float}}$$

Gracias a la propiedad de subsunción podemos reemplazar **Int** por **Float** en la derivación de los tipos conservando la propiedad de ser una expresión válida.

La relación de subtipado puede ser interpretada principalmente de dos formas:

### Interpretación por subconjuntos

Sí  $\mathbf{S} \leq \mathbf{T}$  entonces toda expresión de tipo **S** también es una expresión de tipo **T** pues **S** está contenido en **T**.

### Interpretación por Coerción

Entendemos coerción como la acción de ejercer presión sobre un objeto para forzar su conducta. Esta acción aplicada al contexto de software nos permite modelar el siguiente comportamiento: si  $s$  es de tipo **S** y  $\mathbf{S} \leq \mathbf{T}$  entonces  $s$  se puede convertir de forma única a una expresión de tipo **T**, es decir forzamos el cambio de tipo mediante una conversión explícita.

En el ejemplo anterior teníamos  $2 : \text{Int}$  el cuál fue coercido a  $2.0 : \text{Float}$ .

## 2.1 Subtipado de tipos primitivos

La reglas para definir la relación de subtipado entre los tipos primitivos proporcionados en la definición de los lenguajes de programación constituyen los axiomas de dicho sistema y con ellos se pude derivar el resto de las reglas utilizando las propiedades de subsunción y transitividad.

Para ilustrar este punto se agrega el tipo **Float** junto con los axiomas:

$$\frac{}{\text{Nat} \leq \text{Int}} \qquad \frac{}{\text{Int} \leq \text{Float}}$$

## 2.2 Subtipado de funciones

Definamos el siguiente tipo para una función  $f$  como  $f : Int \rightarrow Int$ , entonces se tiene que  $f n : Int$  cuando recibe cualquier argumento de tipo  $n : Int$ , así  $f n : Float$  por la regla de subsunción que revisamos en la sección anterior, por consiguiente se tiene que:

$$Int \rightarrow Int \leq Int \rightarrow Float$$

es una derivación de tipo válida. Es decir la relación de subtipado original se preserva en el codominio de la función. En tal caso se dice que esta posición es covariante.

Por otro lado si definimos el tipo de nuestra función como  $f : Float \rightarrow Int$ . Tenemos que como todas las expresiones  $e : \mathbf{Int}$  son también de tipo **Float** podemos utilizar la relación de subtipado “en sentido opuesto” para restringir el dominio (de forma inversa a como se usó en el caso anterior), así podemos introducir elementos de tipo **Int** en el dominio de  $f$ . Por lo que se cumple la siguiente relación de subtipado:

$$Float \rightarrow Int \leq Int \rightarrow Int$$

Es decir, la relación de subtipos original  $Int \leq Float$  usada en la primera derivación se invierte para este caso, decimos entonces que este argumento es contravariante.

Finalmente, usando la propiedad de transitividad tenemos la siguiente relación de subtipado

$$Float \rightarrow Int \leq Int \rightarrow Int \leq Int \rightarrow Float$$

De esta forma podemos definir la regla de subtipado para el caso general de los tipos función como sigue:

$$\frac{T_1 \leq S_1 \quad S_2 \leq T_2}{S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2}$$

## 2.3 Subtipado para suma y producto

Para esta aplicación la relación de subtipado se comporta como contravariante, es decir la dirección de la relación se preserva como es de esperar, de tal forma que se introducen las siguientes reglas al sistema de tipos.

$$\frac{S_1 \leq T_1 \quad S_2 \leq T_2}{S_1 + S_2 \leq T_1 + T_2}$$

$$\frac{S_1 \leq T_1 \quad S_2 \leq T_2}{S_1 \times S_2 \leq T_1 \times T_2}$$

## 2.4 Subtipado para registros

Los registros como hemos estudiado en secciones anteriores son la generalización de una **tupla** permitiendo entender el concepto para  $n$  elementos, asociados a éstos existen las **proyecciones** para recuperar el  $n$ -ésimo elemento contenido en ésta.

Veremos brevemente las características asociadas al tipo de esta estructura una vez se introduce la relación de subtipado.

**Amplitud** Dados dos registros A y B donde A posee una cantidad menor de elementos que B, decimos que el tipo del registro A es subtipo del registro B representado de la siguiente forma:

$$\overline{(l_1 : 1, \dots, l_{n+k} : n+k) \leq (l_1 : 1, \dots, l_n : n)}$$

**Profundidad** Esta propiedad es la aplicación de la contravarianza de la relación de subtipo, aplicándose a cada uno de los campos contenidos en el registro:

$$\frac{T_1 \leq S_1 \quad \dots \quad T_n \leq S_n}{(l_1 : T_1, \dots, l_n : T_n) \leq (l_1 : S_1, \dots, l_n : S_n)}$$

**Permutación** El orden de los campos de un valor de tipo registro no importa.

$$\frac{(s_1 : S_1, \dots, s_n : S_n) \text{ permutación de } (l_1 : T_1, \dots, l_n : T_n)}{(s_1 : S_1, \dots, s_n : S_n) \leq (l_1 : T_1, \dots, l_n : T_n)}$$

## 2.5 Elementos máximos para el sistema de tipos

**Top:** En nuestra implementación para el sistema de tipos es importante contar con un elemento máximo, es decir un tipo que cumpla la condición: todo tipo distinto a este subtipo máximo es subtipo del mismo. Es aquí cuando introducimos **Top** que cumple exactamente con esta característica representada de la siguiente forma:

$$\overline{T \leq Top}$$

La idea detrás de **Top** es que a este tipo pertenecen todos aquellos programas que están correctamente tipados. En Java este tipo corresponde al tipo **Object**.

**Bot:** La idea de la existencia de un tipo que es subtipo de todos los demás también existe y se conoce como **Bot**, este tipo es un tipo inhabitado, ninguna expresión debe de existir dentro del mismo. Este tipo ayuda a expresar aquellas expresiones que no deben de regresar ningún valor similar a la forma en la que se utiliza el tipo **Void** en Java.

La descripción anteriormente dada para **Bot** se representa de la siguiente forma:

$$\overline{Bot \leq T}$$

## 3 Casting

En lenguajes de programación como Java hemos hecho uso de este mecanismo para cambiar el tipo de un dato o variable por otro.

Similar al ejemplo que tenemos para pasar de `2:Int` a `2.0:Float`. La relación de subtipado nos provee la regla de subsunción sin embargo definiremos el mecanismo para ambos sentidos de la relación conocidos como **Upcasting** y **Downcasting**

### 3.1 *Upcasting*

En este tipo de casting (también conocido como “casting hacia arriba”) el objetivo es que de un término dado se le atribuya un supertipo del tipo esperado. De esta forma en presencia de subtipos si se tiene  $T \leq R$  entonces podemos concluir, usando subsunción que  $\Gamma \vdash \langle R \rangle e : R$ , en donde  $\langle R \rangle$  es la notación para el *casting* explícito. representado como:

$$\frac{\Gamma \vdash e : T \quad T \leq R}{\Gamma \vdash \langle R \rangle e : R}$$

### 3.2 *Downcasting*

En este tipo de casting (también conocido como “casting hacia abajo”) la asignación de tipos es arbitraria, la relación de subtipado no necesariamente se tiene que cumplir.

Ésto supone una amenaza a la consistencia de tipos en el programa que se escribe con esta instrucción, el lenguaje está forzado a aceptar el downcasting en tiempo de compilación pero en tiempo de ejecución un futuro conjunto de chequeos deben de ser ejecutados para verificar que el tipo asignado no supone un error. Esta regla se representa de la siguiente manera:

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \langle R \rangle e : R}$$

El primer paso es verificar que  $e$  esté correctamente tipada y el segundo es asignar el tipo sin ninguna restricción.

## 4 Ejercicios para el lector

**Ejercicio 4.1.** Dada la siguiente expresión utiliza las reglas de tipado para suma y producto para derivar el tipado de la misma.

$$(4,0 \times 1) - 3,0 : \mathbf{Float}$$

**Ejercicio 4.2.** Dada la siguiente expresión utiliza las reglas de tipado para suma y producto para derivar el tipado de la misma.

$$(3 \times 3,0) + (4 \times 4) : \mathbf{Float}$$

**Ejercicio 4.3.** Dada la siguiente expresión utiliza las reglas de subtipado para funciones para cambiar la firma de la función y que reciba **Float**.

$$fun(x : \mathbf{Int} \rightarrow x + 1) : \mathbf{Int} \rightarrow \mathbf{Int}$$

**Ejercicio 4.4.** Utiliza la regla de **Upcasting** para obtener el tipo de la siguiente expresión basado en las reglas para tipos primitivos.

$$fun(x : \mathbf{Int} \rightarrow if(x \leq 0 \text{ then } (x - 1,0) \text{ else } x)) : \mathbf{Int} \rightarrow \mathbf{Int}$$

**Ejercicio 4.5.** Utiliza las reglas de elementos máximos para obtener la derivación del tipado para la siguiente expresión.

$$fun(x : \mathbf{Int} \rightarrow if(x \leq 0 \text{ then } (x - 1,0) \text{ else } x)) : \mathbf{Bot}$$

## Capítulo 11

# Java Peso Pluma



El presente capítulo constituye el final de este manual. Aquí proporcionaremos la última implementación que revisaremos de un lenguaje orientado a objetos para estudiar las propiedades listadas en el capítulo 10: **Herencia y Subtipado**. Para ello vamos a aproximar una definición al lenguaje orientado a objetos al que más atención hemos prestado a lo largo de materias como **Introducción a Ciencias de la Computación: Java**.

Estudiar lenguajes de programación que no pertenecen al paradigma funcional es una tarea no trivial, dado que la compaginación entre la sintaxis, semántica y manejo de la memoria no es intuitiva con en el caso de **MinHs** donde el mapeo es uno a uno entre la definición y la implementación

## Objetivo

Para propósitos de este manual tomaremos una acercamiento reducido a la implementación de **Java** conservando sus características mas importantes, esta implementación se conoce como: **Java Peso Pluma**.

Nuestro principal interés será entonces la definición de la sintaxis, semántica estática y semántica dinámica de **Java Peso Pluma**.<sup>1</sup>

## Planteamiento

Siguiendo la misma estructura de los capítulos anteriores comenzaremos el estudio de **Java Peso Pluma** proporcionando la definición para la sintaxis concreta de los programas que escribiremos en este lenguaje, la semántica dinámica para evaluarlos y la semántica estática para explorar el sistema de tipos y su relación con la **herencia, métodos y subtipos**. Por último se estudiará brevemente la seguridad de este lenguaje.

# 1 Sintaxis de Java Peso Pluma

A continuación presentamos la tabla con la definición de la sintaxis concreta para este lenguaje. Notemos que en esta definición se encapsulan las principales características de interés de los lenguajes orientados a objetos: clases, objetos, métodos, atributos, herencia, polimorfismo y subtipado junto con la recursión abierta.

**Definición 1.1** (Sintaxis de ). Se presenta la sintaxis del lenguaje separado en categorías y usando las siguientes metavariables:

- Nombres de variables:  $x, y, z$
- Nombres de atributos:  $f, g$
- Nombres de clases:  $C, A, B$
- Nombres de métodos:  $m$

### Expresiones

$e ::= x$	Variables
$e.f$	Acceso a atributo
$e.m(\vec{e})$	Invocación de método
$\text{new } C(\vec{e})$	Instancia de objeto
$(C) e$	Casting

<sup>1</sup>Las definiciones que estudiaremos en este capítulo fueron extraídas de: Enríquez Mendoza J., Lenguajes de Programación Notas de clase: Java Peso Pluma. Universidad Nacional Autónoma de México, 2022.

## Valores

$v ::= \text{new } C(\vec{v})$  Instancia de objeto

## Métodos y Clases

$K ::= C(\vec{C} \vec{x}) \{ \text{super}(\vec{x}); \text{this.} \vec{f} = \vec{x} \}$	Constructores
$M ::= C \vec{m}(\vec{C} \vec{x}) \{ \text{return } e; \}$	Métodos
$L ::= \text{class } C \text{ extends } B \{ \vec{A} \vec{f}; K \vec{M} \}$	Clases

En las definiciones dadas previamente hacemos abuso de notación donde  $\vec{t}$  se conoce como notación vectorial y se interpreta como:

$$\vec{t} =_{def} t_1, t_2, \dots, t_n$$

Represnta entonces una sucesión de n términos.

Cuando se tienen dos vectores sin separación, esta notación se interpreta como el intercalado entre un valor del primero y uno del segundo separando el siguiente par de elementos por una coma:

$$\vec{C} \vec{x} =_{def} C_1 x_1, C_2 x_2, \dots, C_n x_n$$

Para los constructores los vectores de atributo y valores se pueden escribir de la siguiente forma:

$$\text{this.} \vec{f} = \vec{x} =_{def} \text{this.} f_1 = x_1; \text{this.} f_2 = x_2; \dots; \text{this.} f_n = x_n$$

Donde la correspondencia entre cada atributo y valor es uno a uno, de manera implícita denotando la misma cardinalidad en ambos vectores.

Esta definición contempla varias restricciones que enlistaremos a continuación:

- Las clases siempre extienden a una super clase (puede ser **Object**)
- Los constructores se declaran explícitamente.
- Los constructores siempre llaman al constructor de la superclase que se hereda mediante la instrucción **super()**.
- El objeto el cual hace referencia a sus atributos tiene que ser enunciado de forma explícita.
- Los métodos están constituidos únicamente por expresiones **return**.
- Los constructores solo pueden definir al constructor trivial de la clase, recibiendo y asignando un valor por cada atributo de la clase.
- Solo puede existir un único constructor por clase.
- En la definición de clase los atributos solo pueden ser declarados sin asignar ningún valor.

Revisemos la sintáxis concreta de **Java Peso Pluma** con los siguientes ejemplos.

**Ejemplo 1.1.** Proporciona la definición de una clase **TrianguloRec** que modele un triángulo rectángulo y calcule su perímetro (puedes suponer la existencia de la clase **Nat** con la suma implementada).

```
class TrianguloRec extends Object {  
    Nat adyacente;  
    Nat opuesto;  
    Nat hipotenusa;  
  
    Pair (Nat adyacente, Nat opuesto, Nat hipotenusa){  
        super();  
        this.adyacente = adyacente;  
        this.opuesto = opuesto;  
        this.hipotenusa = hipotenusa;  
    }  
  
    Nat perimetro (){  
        return this.adyacente + this.opuesto + this.hipotenusa;  
    }  
}
```

**Ejemplo 1.2.** Proporciona la definición de una clase para definir la dirección de una persona, puedes dar por definida la clase **String**

```
class Direccion extends Object {  
    String calle  
    String colonia  
    String ciudad  
    Nat numero  
    Nat telefono  
  
    Direccion (String calle, String colonia,  
               String ciudad, Nat numero, Nat telefono){  
        super();  
        this.calle = calle;  
        this.colonia = colonia;  
        this.ciudad = ciudad;  
        this.numero = numero  
    }  
}
```

## 2 Tablas de clases

Los programas de **Featherwaight Java** son pares constituidas por una expresión  $e$  y una **tabla de clases**  $T$  cuya generación no es explícita,

Como su nombre indica, esta tabla contiene la información especificada en la declaración de clase, con ella podemos recuperar sus atributos, la firma de los métodos y su cuerpo.

A continuación ilustramos las reglas para operar la **tabla de clase**

**Definición 2.1** (Tabla de Clases). Una tabla de clases es una función finita que asigna clases a nombres de clase, en otras palabras,  $T$  es una sucesión finita de declaraciones de clase de la forma

$$T(C) = \text{class } C \text{ extends } B \{ \vec{A} \vec{f}; K \vec{M} \}$$

### Búsqueda de Atributos

$$\overline{fields(object) = \emptyset}$$

$$\frac{T(C) = \text{class } C \text{ extends } B \{ \vec{C} \vec{f}; K \vec{M} \} \\ (B) = \vec{B} \vec{g}}{(C) = \vec{B} \vec{g}, \vec{C} \vec{f}}$$

### Búsqueda del tipo de un método

$$\frac{T(C) = \text{class } C \text{ extends } D \{ \vec{C} \vec{f}; K \vec{M} \} \\ B \ m(\vec{B}, \vec{x}) \{ \text{return } e; \} \text{ figura en } \vec{M}}{(m, C) = \vec{B} \rightarrow B}$$

$$\frac{T(C) = \text{class } C \text{ extends } D \{ \vec{C} \vec{f}; K \vec{M} \} \\ m \text{ no figura en } \vec{M}}{(m, C) = (m, D)}$$

### Búsqueda del cuerpo de un método

$$\frac{T(C) = \text{class } C \text{ extends } D \{ \vec{C} \vec{f}; K \vec{M} \} \\ B \ m(\vec{B}, \vec{x}) \{ \text{return } e; \} \text{ figura en } \vec{M}}{(m, C) = (\vec{x}, e)}$$

$$\frac{T(C) = \text{class } C \text{ extends } D \{ \vec{C} \vec{f}; K \vec{M} \} \\ m \text{ no figura en } \vec{M}}{(m, C) = (m, D)}$$

### 3 Semántica dinámica

En esta sección vamos a revisar las reglas para poder evaluar expresiones de **Java Peso Pluma**.

La semántica operacional de este lenguaje se representa como:  $\rightarrow_{fj}$  y está basada en la **tabla de clase** para llevar acabo la evaluación. Aunque la utilización de la tabla sea de forma implícita siempre está presente en la aplicación de cada una de las reglas.

**Nota:** En la definición de **Java Peso Pluma** los únicos valores permitidos son aquellos objetos instanciados usando la instrucción **new**, ésto se verá reflejado en la manera en la que las reglas interactúan con dicha instancia de clase.

**Definición 3.1** (Semántica Operacional de **Java Peso Pluma**). Presentamos entonces las reglas de evaluación de **Featherwieght Java**

#### Selección de Atributos

$$\frac{\text{fields}(C) = \vec{C} \vec{f}}{(new C(\vec{v})).f_i \rightarrow_{fj} v_i}$$

#### Invocación de métodos

$$\frac{mbody(m, C) = (\vec{x}, e)}{(new C(\vec{v})).m(\vec{w}) \rightarrow_{fj} e[\vec{x}, \text{this} := \vec{w}, (new C(\vec{v}))]}$$

#### Casting

$$\frac{C \leq D}{(D) (new C(\vec{v})) \rightarrow_{fj} new C(\vec{v})}$$

**Reglas de congruencia** Siguiendo el estándar de evaluación de derecha a izquierda tenemos las siguientes reglas de evaluación para expresiones:

$$\begin{array}{c} \frac{e \rightarrow_{fj} e'}{e.f \rightarrow_{fj} e'.f} \quad \frac{e_i \rightarrow_{fj} e'_i}{e.m(\dots, e_i, \dots) \rightarrow_{fj} e.m(\dots, e'_i, \dots)} \\ \frac{e \rightarrow_{fj} e'}{e.m(\vec{e}) \rightarrow_{fj} e'.m(\vec{e})} \quad \frac{e_i \rightarrow_{fj} e'_i}{new C(\dots, e_i, \dots) \rightarrow_{fj} new C(\dots, e'_i, \dots)} \\ \frac{e \rightarrow_{fj} e'}{(C) e \rightarrow_{fj} (C) e'} \end{array}$$

## 4 Semántica estática

El sistema de tipos de **Java Peso Pluma** no cuenta con tipos primitivos, es decir que todos los tipos con los que trabajemos tendrán que ser definidos mediante su respectiva clase.

En este lenguaje el tipo **Top** será Object y este no tendrá que ser definido pues representa el tipo más primitivo con el cual podemos definir construcciones más complejas, este constituye también una palabra reservada en la definición que propusimos de la sintaxis.

**Definición 4.1** (Semántica Estática de **Java Peso Pluma**). A continuación enunciamos las reglas de tipado

### Reglas de Subtipado

$$\frac{T(C) = \text{class } C \text{ extends } D \{ \vec{C} \vec{f}; K \vec{M} \}}{C \leq D}$$

Es decir, si la clase **C** extiende a la clase **D** entonces **C** es subtipo de **D**.

En este lenguaje las propiedades de **reflexividad** y **transitividad** deben de mantenerse aplicables.

### Reglas de Tipado

$$\frac{}{\Gamma, x : C \vdash x : C} \quad \text{Variables}$$

$$\frac{\Gamma \vdash e : C \quad fields(C) = \vec{C} \vec{f}}{\Gamma \vdash e.f_1 : C_i} \quad \text{Acceso a atributos}$$

$$\frac{\Gamma \vdash e_0 : C_0 \quad \Gamma \vdash \vec{e} : \vec{C} \quad mtype(m, C_0) = \vec{D} \rightarrow C \quad \vec{C} \leq \vec{D}}{\Gamma \vdash e_0.m(\vec{e}) : C} \quad \text{Invocación de métodos}$$

$$\frac{\Gamma \vdash \vec{e} : \vec{C} \quad fields(C) = \vec{D} \vec{f} \quad \vec{C} \leq \vec{D}}{\Gamma \vdash \text{new } C(\vec{e}) : C} \quad \text{Creación de objetos}$$

### Reglas de casting

$$\frac{\Gamma \vdash e : D \quad D \leq C}{\Gamma \vdash (C)e : C} \quad \text{Upcasting}$$

$$\frac{\Gamma \vdash e : D \quad C \leq D \quad C \neq D}{\Gamma \vdash (C)e : C} \quad \text{Downcasting}$$

$$\frac{\Gamma \vdash e : D \quad C \not\leq D \quad D \not\leq C \quad \text{stupid warning}}{\Gamma \vdash (C)e : C} \quad \text{Casting estúpido}$$

Esta última regla es un tecnicismo necesario para poder probar la preservación de tipos con la semántica operacional estructural.

**Formación de Clases** Introducimos tres juicios para denotar la correcta formación de una clase:  $M \text{ ok in } C$  para indicar que el método  $M$  está bien formado en la clase  $C$ ,  $C \text{ ok}$  que indica que la clase  $C$  está bien formada y el juicio  $T \text{ ok}$  para decir que la tabla de clases  $T$  está bien formada.

$$\frac{\begin{array}{c} K = C(\vec{D}\vec{y}, \vec{C}\vec{x}) \{ \text{super}(\vec{y}); \text{this.}f = \vec{x} \} \\ \text{fields}(D) = \vec{D}\vec{g} \\ \vec{M} \text{ ok in } C \end{array}}{\text{class } C \text{ extends } D \{ \vec{C}\vec{f}; K \vec{M} \} \text{ ok}}$$

### Formación de métodos

$$\frac{\begin{array}{c} T(C) = \text{class } C \text{ extends } D \{ \dots \} \\ mtype(m, D) = \vec{C} \rightarrow C_0 \\ \vec{x} : \vec{C}, \text{this} : C \vdash e : C'_0 \\ C'_0 \leq C_0 \end{array}}{C_0 \text{ m } (\vec{C}\vec{x}) \{ \text{return } e; \} \text{ ok in } C}$$

**Formación de Tablas** Decimos que una tabla de clases está bien formada si todas las clases declaradas en ella están bien formadas, modelado con la regla:

$$\frac{\forall C \in \text{dom}(T), T(C) \text{ ok}}{T \text{ ok}}$$

## 5 Propiedades de Java Peso Pluma

Como hemos acostumbrado en el estudio de cada uno de los lenguajes definidos en este manual brevemente mencionaremos las dos propiedades que más interesan en este curso: **progreso de la función**  $\rightarrow_{fj}$  y **preservación de tipos**.

### 5.1 Preservación de tipos

La preservación de tipos que hemos estudiado en implementaciones anteriores difiere en que ahora tenemos que considerar su relación con el mecanismo de herencia y los castings que no son correctos. Esto se puede observar en la siguiente regla.

**Proposición 5.1** (Preservación de tipos). Si  $T$  es una tabla de clases bien formada,  $\Gamma \vdash e : C$  y  $e \rightarrow_{fj} e'$ , entonces existe  $C'$  tal que  $C' \leq C$  y  $\Gamma \vdash e' : C'$

### 5.2 Progreso

Esta propiedad tiene un pequeño detalle cuando se utiliza dentro del contexto de **Java Peso Pluma**. En general la propiedad es válida salvo en el caso único en el que no se puede computar un downcasting.

**Proposición 5.2** (Progreso de la relación  $\rightarrow_{fj}$ ). Sea  $T$  una tabla de clases bien formada, si  $\emptyset \vdash e : C$  entonces sucede una y solo una de las siguientes condiciones:

- $e$  es un valor.
- $e$  contiene una expresión de la forma  $(C) \text{ new } D(\vec{v})$  en donde  $D \leq C$ , es decir, no se puede realizar el *downcast*.
- Existe  $e'$  tal que  $e \rightarrow_{fj} e'$ .

### 5.3 Seguridad

Combinando las dos propiedades anteriores obtenemos la regla de seguridad, esta enuncia que si dada una expresión  $e$  de **Java Peso Pluma** se obtiene una expresión en su forma normal, o bien es un valor o encontramos un *downcasting* incorrecto.

**Proposición 5.3** (Seguridad de **Java Peso Pluma**). Si  $T$  es una tabla de clases bien formada,  $\emptyset \vdash e : C$  y  $e \rightarrow_{fj}^* e'$  con  $e'$  en forma normal, entonces se cumple una y solo una de las siguientes condiciones:

- $e'$  es un valor  $v$  tal que  $\emptyset \vdash v : D$  y  $D \leq C$ .
- $e'$  contiene como subexpresión  $(C) \text{ new } D(\vec{v})$  en donde  $D \leq C$ .

Dado cualquier expresión  $e$ , o bien esta eventualmente llega a ser evaluada como un valor o se bloquea en un *downcasting* que no se puede resolver.

## 6 Cómo se relaciona Java Peso Pluma con Java?

**Java Peso Pluma** es un lenguaje con propósitos ilustrativos, muchas de las características que rebotucen a **Java** y que son englobadas por el paradigma de la **OOP**<sup>2</sup> se dejan fuera del enfoque de la definición del mismo. Es natural entonces preguntarse por la relación que guardan ambos lenguajes, qué sucede cuando escribimos un lenguaje en **Java Peso Pluma** y lo queremos ejecutar en la **JVM**<sup>3</sup>.

Para dar respuesta a esta pregunta presentamos las siguientes proposiciones que hacen explícita la correspondencia entre ambos lenguajes.

**Proposición 6.1.** Cada programa sintácticamente correcto en **Java Peso Pluma** es también sintácticamente correcto en **Java**.

**Proposición 6.2.** Un programa sintácticamente correcto es tipable en **Java Peso Pluma** si y sólo si es tipable en **Java**.

<sup>2</sup>Object Oriented Programming por sus siglas en inglés.

<sup>3</sup>Java Virtual Machine.

**Proposición 6.3.** La ejecución de un programa bien tipado en **Java Peso Pluma** se comporta de la misma forma en **Java**.

**Proposición 6.4.** La evaluación de un programa en **Java Peso Pluma** no termina si y sólo si compilarlo y ejecutarlo en **Java** causa no terminación.

La demostración de estas últimas cuatro proposiciones no es posible dado que no hay una formalización de **Java** que nos permita razonar sobre si mismo, sin embargo enumerar estas proposiciones ilustra la importancia de estudiar los lenguajes de programación formalmente para inferir propiedades y características deseadas bajo un modelo de razonamiento lógico.

## 7 Ejercicios para el lector

**Ejercicio 7.1.** Dada la definición de **Java Peso Pluma** proporciona un programa para modelar árboles binarios cuyo contenido de sus nodos sean enteros (Puedes dar por definido el tipo **Int**).

**Ejercicio 7.2.** Dada la definición de **Java Peso Pluma** proporciona un programa para modelar un nuevo objeto llamado **Empleado** que contenga un campo de clase para su dirección, su número de seguridad social (NSS) y su salario (puedes dar por definidas las clases **Int**, **String** y **Direccion** del ejemplo 1.2).

**Ejercicio 7.3.** Dadas las reglas para la sintáxis y las reglas de la semántica dinámica y estática de **Featherwieght Java**, Demuestra la proposición 5.1: **Preservación de tipos**.

**Ejercicio 7.4.** Dadas las reglas para sintáxis y las reglas de la semántica dinámica y estática de **Featherwieght Java**, Demuestra la proposición 5.2: **Progreso**.

**Ejercicio 7.5.** Dadas las reglas para sintáxis y las reglas de la semántica dinámica y estática de **Featherwieght Java**, Demuestra la proposición 5.3: **Seguridad**.

# Bibliografía

- [1] Ramírez Pulido K., Soto Romero M., Enríquez Mendoza J., Nota de Clase del curso de Lenguajes de Programación, Facultad de Ciencias UNAM, 2022.
- [2] Brooks A., Modern Programming Languages: A Practical Introduction (2nd Edition). Sherwood, Oregon : Franklin, Beedle & Associates, cop. 2011.
- [3] Deepika P., Selection Sort (2021), <https://www.geeksforgeeks.org/selection-sort/> Acceso: Noviembre 14, 2022.
- [4] Lipovaca M., Learn You a Haskell for Great Good!: A Beginner's Guide. 401 China Basin Street Suite 108 San Francisco, CA. United States. 2011.
- [5] Miranda Perea F., González Huesca L., Nota de Clase del curso de Lenguajes de Programación, Facultad de Ciencias UNAM. 2021.
- [6] Keller G., O'Connor-Davis L., Class Notes from the course Concepts of programming language design, Department of Information and Computing Sciences, Utrecht University, The Netherlands, 2020.
- [7] Nielson F., Semantics with Applications: An Appetizer, Springer Publishing, 2007.
- [8] Harper R., Practical Foundations for Programming Languages. Working draft, 2010.
- [9] Mitchell J., Foundations for Programming Languages. MIT Press, 1996.
- [10] Krishnamurthi S., Programming Languages Application and Interpretation; Version 26.04.2007.
- [11] Spector-Zabusky A., How would the Lambda Calculus add numbers? (2021). <https://stackoverflow.com/questions/29756732/how-would-the-lambda-calculus-add-numbers>. Acceso: Abril 16, 2023.
- [12] Enríquez Mendoza J., Lenguajes de Programación Nota de clase. Universidad Nacional Autónoma de México. 2022.
- [13] Keller. G., O'Connor-Davis. L., Concepts of Programming Languages: Data types in Explicitly Typed Languages, 2022.

- [14] CS4114 Formal Languages Spring Chapter 1 Introduction Grammar Exercises (2021), <https://opendsa-server.cs.vt.edu/OpenDSA/Books/PIFLAS21/html/IntroGrammarEx.html>. Acceso: Septiembre 7, 2022.
- [15] Binary Search, a haskell approach (2022), <https://programming-idioms.org/idiom/124/binary-search-for-a-value-in-sorted-array/2120/haskell>. Acceso: Septiembre 29. 2022.
- [16] Binary Search Data Structure and Algorithm Tutorials (2023), <https://www.geeksforgeeks.org/binary-search/>. Acceso: Septiembre 29. 2022.
- [17] Dahiya A., Wasserman Z., CIS 194: Introduction to Haskell, Homework 1 (2013), <https://www.seas.upenn.edu/cis1940/spring13/hw/01-intro.pdf>. Acceso: Noviembre 4, 2022.
- [18] Yorgey B., Typeclassopedia (2011), <https://wiki.haskell.org>Typeclassopedia>. Acceso: Noviembre 24, 2022.
- [19] King J. K., Haskell List Problem Set (2018), <https://github.com/JD95/haskell-problem-sets/blob/master/Lists/Problems.hs>. Acceso: Diciembre 10, 2022.
- [20] Goguen, Joseph A. (1975). Semantics of computation. Category Theory Applied to Computation and Control. Lecture Notes in Computer Science. Vol. 25. Springer. pp. 151–163.
- [21] Floyd, Robert W. (1967). Assigning Meanings to Programs. In Schwartz, J.T. (ed.). Mathematical Aspects of Computer Science. Proceedings of Symposium on Applied Mathematics. Vol. 19. American Mathematical Society. pp. 19–32.
- [22] Winskel, Glynn (1993). The formal semantics of programming languages: an introduction. Cambridge, Mass.: MIT Press. p. xv
- [23] Schmidt, David A. (1986). Denotational Semantics: A Methodology for Language Development. William C. Brown Publishers.
- [24] Plotkin, Gordon D. (1981). A structural approach to operational semantics (Report). Technical Report DAIMI FN-19. Computer Science Department, Aarhus University.
- [25] Deransart, Pierre; Jourdan, Martin; Lorho, Bernard (1988). Attribute Grammars: Definitions, Systems and Bibliography. Lecture Notes in Computer Science 323.
- [26] Krishnamurthi, Shriram (2012). Programming Languages: Application and Interpretation (2nd ed.).
- [27] Slonneger, Kenneth; Kurtz, Barry L. (1995). Formal Syntax and Semantics of Programming Languages. Addison-Wesley.
- [28] Colaboradores de Wikipedia. Semantics (computer science). Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 18 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Semantics\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Semantics_(computer_science)).

- [29] Colaboradores de Wikipedia. Syntax (programming languages). Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 18 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Syntax\\_\(programming\\_languages\)](https://en.wikipedia.org/wiki/Syntax_(programming_languages)).
- [30] Friedman, Daniel P.; Mitchell Wand; Christopher T. Haynes (1992). Essentials of Programming Languages (1st ed.). The MIT Press.
- [31] Smith, Dennis (1999). Designing Maintainable Software. Springer Science Business Media.
- [32] Aho, Alfred V.; Monica S. Lam; Ravi Sethi; Jeffrey D. Ullman (2007). Compilers: Principles, Techniques, and Tools (2nd ed.). Addison Wesley.
- [33] Louden, Kenneth C. (1997). Compiler Construction: Principles and Practice. Brooks/Cole. Exercise 1.3, pp.27–28.
- [34] Michael Sipser (1997). Introduction to the Theory of Computation. PWS Publishing. Section 2.2: Pushdown Automata, pp.101–114
- [35] Colaboradores de Wikipedia. Pragmatics. Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 19 10 2023]. Disponible en <https://en.wikipedia.org/wiki/Pragmatics>.
- [36] Coppock, Elizabeth; Champollion, Lucas (2019). Invitation to Formal Semantics (PDF). Manuscript. p. 37.
- [37] Mey, Jacob L. (1993) Pragmatics: An Introduction. Oxford: Blackwell (2nd ed. 2001).
- [38] Shriram Krishnamurthi, David Tucker, Notas del curso de Lenguajes en la Universidad de Brown, Primavera-2007
- [39] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, How to Design Programs, First Edition, The MIT Press
- [40] Colaboradores de Wikipedia. Ambiguous grammar. Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 19 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Ambiguous\\_grammar](https://en.wikipedia.org/wiki/Ambiguous_grammar)
- [41] Willem J. M. Levelt (2008). An Introduction to the Theory of Formal Languages and Automata. John Benjamins Publishing.
- [42] Scott, Elizabeth (April 1, 2008). "SPPF-Style Parsing From Earley Recognizers". Electronic Notes in Theoretical Computer Science. 203 (2): 53–67.
- [43] Colaboradores de Wikipedia. Compiled Language. Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 19 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Compiled\\_language](https://en.wikipedia.org/wiki/Compiled_language)
- [44] Colaboradores de Wikipedia. Interpreter (computing). Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 19 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Interpreter\\_\(computing\)](https://en.wikipedia.org/wiki/Interpreter_(computing))

- [45] Terence Parr, Johannes Luber, The Difference Between Compilers and Interpreters Archived 2014-01-06 at the Wayback Machine
- [46] Compilers vs. interpreters: explanation and differences. IONOS Digital Guide. Retrieved 2022-09-16.
- [47] Colaboradores de Wikipedia. Object-oriented programming. Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 19 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Object-oriented\\_programming](https://en.wikipedia.org/wiki/Object-oriented_programming)
- [48] Abadi, Martin; Luca Cardelli (1998). A Theory of Objects. Springer Verlag.
- [49] Armstrong, Deborah J. (February 2006). "The Quarks of Object-Oriented Development". Communications of the ACM. 49 (2): 123–128.
- [50] Colaboradores de Wikipedia. Programación por procedimientos. Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 19 10 2023]. Disponible en [https://es.wikipedia.org/wiki/Programacion\\_por\\_procedimientos](https://es.wikipedia.org/wiki/Programacion_por_procedimientos)
- [51] Colaboradores de Wikipedia. Functional programming. Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 23 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Functional\\_programming](https://en.wikipedia.org/wiki/Functional_programming).
- [52] Hudak, Paul (September 1989). Conception, evolution, and application of functional programming languages. ACM Computing Surveys. 21 (3): 359–411.
- [53] Jain, Anisha (2022-12-10). Javascript Promises— Is There a Better Approach?. Medium. Retrieved 2022-12-20.
- [54] Colaboradores de Wikipedia. Imperative programming. Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 23 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Imperative\\_programming](https://en.wikipedia.org/wiki/Imperative_programming)
- [55] Imperative programming: Overview of the oldest programming paradigm. IONOS Digitalguide. 21 May 2021. Retrieved 2022-05-03.
- [56] Bruce Eckel (2006). Thinking in Java. Pearson Education. p. 24.
- [57] Colaboradores de Wikipedia. Logic programming. Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 24 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Logic\\_programming](https://en.wikipedia.org/wiki/Logic_programming)
- [58] Colaboradores de Wikipedia. Mathematical object. Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 25 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Mathematical\\_object](https://en.wikipedia.org/wiki/Mathematical_object)
- [59] Azzouni, J., 1994. Metaphysical Myths, Mathematical Practice. Cambridge University Press.
- [60] Burgess, John, and Rosen, Gideon, 1997. A Subject with No Object. Oxford Univ. Press.

- [61] Colaboradores de Wikipedia. Judgment (mathematical logic). Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 25 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Judgment\\_\(mathematical\\_logic\)](https://en.wikipedia.org/wiki/Judgment_(mathematical_logic))
- [62] Martin-Löf, Per (1996). On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*. 1 (1): 11–60
- [63] Colaboradores de Wikipedia. Rule of inference. Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 25 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Rule\\_of\\_inference](https://en.wikipedia.org/wiki/Rule_of_inference)
- [64] Boolos, George; Burgess, John; Jeffrey, Richard C. (2007). Computability and logic. Cambridge: Cambridge University Press. p. 364
- [65] John C. Reynolds (2009) [1998]. Theories of Programming Languages. Cambridge University Press. p. 12
- [66] Bergmann, Merrie (2008). An introduction to many-valued and fuzzy logic: semantics, algebras, and derivation systems. Cambridge University Press. p. 100
- [67] Miranda Perea F., Elisa Viso G., Matemáticas Discretas, Facultad de Ciencias UNAM. 2016. pp 163-92.
- [68] John A. Dossey, Albert D. Otto, Lawrence E. Spence, and Charles Vanden Eynden. Discrete Mathematics. Pearson/Addison-Wesley, 5-th edition, 2006
- [69] Judith L. Gersting. Mathematical Structures for Computer Science. Computer Science Press, W.H. Freeman and Company, third edition, 1993.
- [70] Winfried Karl Grassman and Jean-Paul Tremblay. LOGIC AND DISCRETE MATHEMATICS, A Computer Science Perspective. Prentice-Hall Inc., 1996
- [71] David Gries and Fred B. Schneider. A Logical Approach to Discrete Mathematics. Springer-Verlag, 1994.
- [72] Jerold W. Grossman. DISCRETE MATHEMATICS, an introduction to concepts, methods and applications. Macmillan Publishing Company, 1990
- [73] Thomas Koshy. Discrete Mathematics with Applications. Elsevier Academoc Press, 2004
- [74] K.H. Rossen. Discrete Mathematics and its Applications. McGraw Hill, 6-th edition, 2006.
- [75] Y. Brent. A. Adi, W. Zach. CIS 194: Introduction to Haskell. homework 1. University of Pennsylvania, 2013.
- [76] Krahn, H. Rumpe. B, Völke. S. Model Driven Engineering Languages and Systems. Technische Universität Braunschweig, Braunschweig, Germany. 2007. pp 286–300.
- [77] Chomsky N. Aspects of the Theory of Syntax. Massachusetts Institute of Technology Press. 2014.

- [78] Colaboradores de Wikipedia. Parse tree. Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 30 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Parse\\_tree](https://en.wikipedia.org/wiki/Parse_tree)
- [79] Colaboradores de Wikipedia. Abstract syntax. Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 30 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Abstract\\_syntax](https://en.wikipedia.org/wiki/Abstract_syntax)
- [80] Colaboradores de Wikipedia. Scope. Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 31 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Scope\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Scope_(computer_science))
- [81] Colaboradores de Wikipedia. Let expression. Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 31 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Let\\_expression](https://en.wikipedia.org/wiki/Let_expression)
- [82] Colaboradores de Wikipedia. Lambda Calculus. Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 31 10 2023]. Disponible en [https://en.wikipedia.org/wiki/Lambda\\_calculus](https://en.wikipedia.org/wiki/Lambda_calculus)
- [83] Turing, Alan M. (December 1937). "Computability and -Definability". *The Journal of Symbolic Logic*. 2 (4): 153–163
- [84] Coquand, Thierry (8 February 2006). Zalta, Edward N. (ed.). "Type Theory". *The Stanford Encyclopedia of Philosophy* (Summer 2013 ed.). Retrieved November 17, 2020
- [85] Mitchell, John C. (2003). Concepts in Programming Languages. Cambridge University Press. p. 57.
- [86] Pierce, Benjamin C. Basic Category Theory for Computer Scientists. p. 53.
- [87] Church, Alonzo (1932). A set of postulates for the foundation of logic. *Annals of Mathematics. Series 2*. 33 (2): 346–366.
- [88] Selinger, Peter (2008), Lecture Notes on the Lambda Calculus (PDF), vol. 0804, Department of Mathematics and Statistics, University of Ottawa, p. 9
- [89] Turbak, Franklyn; Gifford, David (2008), Design concepts in programming languages, MIT press, p. 251,
- [90] Abrahams, P. W. (1966). A final solution to the Dangling else of ALGOL 60 and related languages. *Communications of the ACM*. 9 (9): 679–682.