# Estructura de la materia a grandes rasgos:

**Primera Parte:** Lenguaje imperativo

**Segunda Parte:** Lenguaje aplicativo puro, y lenguaje aplicativo con referencias y asignación

# Ejes de Contenidos de la primera parte

1. Introducción a la sintaxis y la semántica de lenguajes
2. El problema de dar significado a la recursión e iteración
3. Un Lenguaje Imperativo

# ¿Cómo se otorga significado a un lenguaje?

El establecer el significado de las frases de un lenguaje de programación es un problema de múltiples aristas en tanto puede tener variados objetivos, que van desde la comprensión humana hasta la necesidad de que una máquina los pueda interpretar o trauducir a una secuencia de instrucciones ejecutables.

Del significado trata un manual de usuario, en tanto provee una descripción intuitiva de una acción o una denotación, y también un intérprete, un compilador, o un herramienta teórica destinada a desentrañar principios básicos de diseño.

# Primera aproximación

**Maneras diferentes de dar significado a un lenguaje:**

- *informal, intuitiva*:

  explica el funcionamiento de los programas a través de frases comprensibles en el lenguaje natural (ejemplo: manuales, documentación tipo javadoc)

# Primera aproximación

**Maneras diferentes de dar significado a un lenguaje:**

- *informal, intuitiva*
- *axiomática*:

  explica el sentido dinámico de manera implícita, estableciendo en el marco de una lógica qué propiedades son asignables a una determinada frase del lenguaje, estableciendo así una manera de razonar sobre programas (ejemplo: $\{P/v \leftarrow e\}\ v := e\ \{P\}$)

# Primera aproximación

**Maneras diferentes de dar significado a un lenguaje:**

- *informal, intuitiva*
- *axiomática*
- *operacional*:

  explica el sentido dinámico de manera explícita, diciendo de que manera se ejecuta un programa (ejemplo: intérprete)

# Primera aproximación

**Maneras diferentes de dar significado a un lenguaje:**

- *informal, intuitiva*
- *axiomática*
- *operacional*
- *denotacional*:

  Asigna a cada programa un significado estático en un universo semántico especialmente definido para representar los fenómenos que el lenguaje describe .

# Primera aproximación

**Maneras diferentes de dar significado a un lenguaje:**

- *informal, intuitiva*
- *axiomática*
- *operacional*
- *denotacional*:

  Asigna a cada programa un significado estático en un universo semántico especialmente definido para representar los fenómenos que el lenguaje describe . Este universo podría ser un universo matemático con estructura, o el universo de las frases de otro lenguaje, por ejemplo un compilador.

# Semántica Denotacional

Requiere definir:

- Domino semántico
- Función semámtica

$L$ = conjunto de frases del lenguaje
$D$ = dominio semántico formado por objetos abstractos
$[\![ \_ ]\!]$ = función semántica

$$L \xrightarrow{[\![ \_ ]\!]} D$$

$$e \longrightarrow [\![e]\!]$$

# Dominio semántico, función semántica

*BIN* = secuencias finitas de ceros y unos

$$BIN = \{\alpha_0 ... \alpha_{n-1} : n \geq 1 \text{ y } \alpha_i \in \{0, 1\}\}$$

**D** = los naturales con el cero

$$[\![ \_ ]\!] \ : \ BIN \rightarrow \mathbf{D}$$

$$[\![\alpha_0 ... \alpha_{n-1}]\!] = \sum_{i=1}^{n} \alpha_{i-1} 2^{n-i}$$

# Problemas que requieren nuevas herramientas

- El problema de dar la sintaxis del lenguaje, es decir una determinación del conjunto de frases que serán consideradas programas válidos
- El problema de la buena definición del dominio semántico y la función semántica
- El manejo de variables, la ligadura y los problemas de captura
- El problema del significado de la recursión

# Gramáticas

Para el estudio de los lenguajes en general, las gramáticas convencionales no son una herramienta adecuada porque sufren del problema de la *ambiguedad*.

$$\begin{aligned} \langle intexp \rangle \quad ::= \quad & 0 \mid 1 \mid 2 \mid ... \\ & -\langle intexp \rangle \mid \\ & \langle intexp \rangle + \langle intexp \rangle \mid \\ & \langle intexp \rangle * \langle intexp \rangle \end{aligned}$$

Observemos ejemplos de frases generadas por esta gramática:

-15
-15+3
2*-4

# Ambiguedad

La gramátíca es ambigua en el siguiente sentido: algunas frases admiten diferentes maneras de generarse. Tal es el caso de 2+3*4:

(1)

$$
\begin{aligned}
& \langle intexp \rangle \\
\rightarrow\ & \langle intexp \rangle * \langle intexp \rangle \\
\rightarrow\ & \langle intexp \rangle * 4 \\
\rightarrow\ & \langle intexp \rangle + \langle intexp \rangle * 4 \\
\rightarrow\ & \langle intexp \rangle + 3 * 4 \\
\rightarrow\ & 2 + 3 * 4
\end{aligned}
$$

(2)

$$
\begin{aligned}
& \langle intexp \rangle \\
\rightarrow\ & \langle intexp \rangle + \langle intexp \rangle \\
\rightarrow\ & 2 + \langle intexp \rangle \\
\rightarrow\ & 2 + \langle intexp \rangle * \langle intexp \rangle \\
\rightarrow\ & 2 + 3 * \langle intexp \rangle \\
\rightarrow\ & 2 + 3 * 4
\end{aligned}
$$

# Solución que proveen los lenguajes usuales

**Paréntesis y convenciones de precedencia:** la gramática dice cómo se escriben concretamente las frases del lenguaje.

$$
\begin{array}{lcl}
\langle intexp \rangle & ::= & \langle intexp \rangle + \langle termexp \rangle | \\
 & & \langle intexp \rangle * \langle termexp \rangle \mid \langle termexp \rangle \\
\langle termexp \rangle & ::= & \langle groundexp \rangle | - \langle termexp \rangle \\
\langle groundexp \rangle & ::= & 0|1|2|....|(\langle intexp \rangle)
\end{array}
$$

Aquí queda claro que el + asocia a izquierda y que el menos tiene mayor precedencia. Podríamos llamarla **gramática concreta**, y a las frases que genera, **frases concretas**. Podríamos decir que define la **sintaxis concreta** del lenguaje.

# La gramática concreta no es apropiada

- La gramática concreta resulta más complicada que la que dimos anteriormente.
- Nos obliga a fijar detalles de la sintaxis del lenguaje que son irrelevantes para nuestros propósitos, y que de hecho cada lenguaje de programación lo resuelve de una manera distinta

Nos interesa describir sólo la estructura de las frases, decir cuáles son las construcciones que hay en el lenguaje.

No interesa cómo se escriben las frases sino qué frases hay.

# Gramáticas abstractas

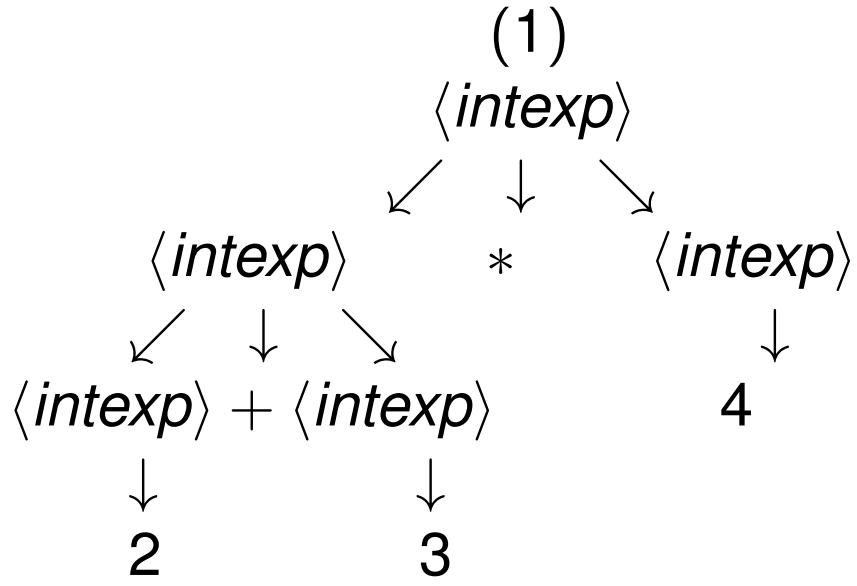
Utilizamos la forma de definición de las gramáticas convencionales:

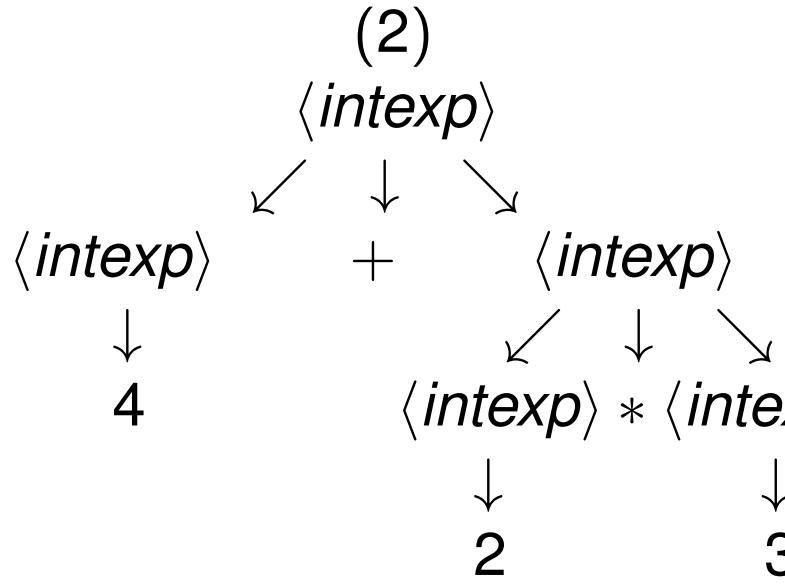
$$\begin{array}{lll} \langle intexp \rangle & ::= & 0 \mid 1 \mid 2 \mid ... \\ & & -\langle intexp \rangle \mid \\ & & \langle intexp \rangle + \langle intexp \rangle \mid \\ & & \langle intexp \rangle * \langle intexp \rangle \end{array}$$

Para nosotros será una gramática abstracta, que expresa qué construcciones tiene el lenguaje, cuál es la estructura de las frases que hay, y cuáles son las subfrases.

# Estructura de las frases abstractas

Estructuras abstractas diferentes se ponen de manifiesto al representar cada derivación a través de un *árbol sintáctico*:



# Trabajaremos con gramáticas abstractas

**Problema**: ¿qué notación que utilizaremos para referirnos a una producción particular de tal gramática?

En las ocasiones en que nos resulte necesario precisar notación específica resolveremos los problemas de ambigüedad mencionados estableciendo precedencias, "stopping symbols" y utilizando los paréntesis que sean necesarios.

Tales símbolos **no serán parte de la gramática** en cuestión, sino sólo convenciones de notación que nos permitirán entender de qué producción de la gramática abstracta estamos hablando.

# Semántica denotacional

Es necesario recurrir a herramientas matemáticas para la construcción de los dominios semánticos y para la definición misma de la función.

- La semántica denotacional debe ser una herramienta que aporte claridad conceptual, y no que sea un instrumento de traducir algo poco comprensible en otra cosa incomprensible,
- la semántica denotacional debe ser útil para comparar otras posibles semánticas,
- se deben poder utilizar resultados típicos de la teoría de lenguajes para estudiar las características del lenguaje en cuestión.

# Función semántica y Ecuaciones semánticas

## Función semántica:

$[\![ \_ ]\!] : \langle intexp \rangle \to \mathbf{Z}$

## Ecuaciones semánticas:

$$
\begin{array}{lcl}
[\![0]\!] & = & 0 \\
[\![1]\!] & = & 1 \\
 & \vdots & \\
[\![-e]\!] & = & -[\![e]\!] \\
[\![e + e']\!] & = & [\![e]\!] + [\![e']\!] \\
[\![e * e']\!] & = & [\![e]\!] * [\![e']\!]
\end{array}
$$

# Aspectos destacables de la definición

- Los mismos símbolos aparecen con dos sentidos distintos. El 0 de la derecha es una frase del lenguaje de las expresiones enteras, y el de la izquierda es el objeto cero perteneciente a los números enteros. Esta aparente circularidad tiene un nombre: **metacircularidad**.
- Cada una de las tres últimas ecuaciones representa, en realidad, infinitas ecuaciones. Por ejemplo, $[\![-e]\!] = -[\![e]\!]$ establece una propiedad que vale cualquiera sea la expresión *e*. Aquí *e* NO es una expresión, sólo es un objeto que representa cualquier expresión del lenguaje. Es una variable del **metalenguaje**, o sea una **metavariable**.

# Ecuaciones semánticas

¿Todo conjunto de ecuaciones semánticas define una función?

$$
\begin{array}{lcl}
[\![0]\!] & = & 0 \\
[\![e + e']\!] & = & [\![e]\!] + [\![e']\!] \\
[\![e * 0]\!] & = & 0 \\
[\![e * (e' + e'')]\!] & = & [\![e * e']\!] + [\![e * e'']\!] \\
[\![(e + e') * e'')]\!] & = & [\![e * e'']\!] + [\![e' * e'']\!]
\end{array}
$$

$$[\![1 * 1]\!] = ?$$

# Ecuaciones semánticas

Matemáticamente lo resolveríamos agregando una ecuación:

$$
\begin{aligned}
[\![0]\!] &= 0 \\
[\![e + e']\!] &= [\![e]\!] + [\![e']\!] \\
[\![e * 0]\!] &= 0 \\
[\![1 * 1]\!] &= 1 \\
[\![e * (e' + e'')]\!] &= [\![e * e']\!] + [\![e * e'']\!] \\
[\![(e + e') * e'')]\!] &= [\![e * e'']\!] + [\![e' * e'']\!]
\end{aligned}
$$

$$
[\![2 * 2]\!] = [\![2 * (1 + 1)]\!] = [\![2 * 1]\!] + [\![2 * 1]\!] =
$$

$$
[\![((1 + 1) * 1]\!] + [\![(1 + 1) * 1]\!] = \ldots
$$

# Ecuaciones semánticas

Dado un conjunto de ecuaciones semánticas es en general muy difícil determinar si el conjunto define realmente una (única) función en el conjunto de expresiones del lenguaje.

A la hora de definir la semántica de un lenguaje de programación tenemos que poder garantizar que las ecuaciones semánticas dadas definen una semántica.

# Dirección por sintaxis

Un conjunto de ecuaciones es *dirigido por sintaxis* cuando se satisfacen las siguientes condiciones:

- hay 1 ecuación por cada producción de la gramática abstracta
- cada ecuación que expresa el significado de una frase compuesta, lo hace puramente en función de los significados de sus subfrases inmediatas

# Composicionalidad

La dirección por sintaxis garantiza existencia y unicidad del significado, pero además garantiza la propiedad de **composicionalidad de una función.**

Se dice que una semántica es **composicional**, cuando el significado de una frase no depende de ninguna propiedad de sus subfrases, salvo de sus significados.

**Efectos de la composicionalidad**: podemos reemplazar una subfrase $e_0$ de $e$ por otra de igual significado que $e_0$ sin alterar el significado de la frase $e$.

# Los conceptos de variable y ligadura

## Gramática abstracta para el lenguaje de los predicados

$$
\begin{array}{rcl}
\langle intexp \rangle & ::= & 0 \mid 1 \mid 2 \mid \ldots \\
& & \langle var \rangle \\
& & -\langle intexp \rangle \\
& & \langle intexp \rangle + \langle intexp \rangle \\
& & \langle intexp \rangle * \langle intexp \rangle \\
& & \langle intexp \rangle - \langle intexp \rangle \\
& & \langle intexp \rangle \,/\, \langle intexp \rangle \\
& & \langle intexp \rangle \,\%\, \langle intexp \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle assert \rangle & ::= & \textbf{true} \mid \textbf{false} \\
& & \langle intexp \rangle = \langle intexp \rangle \\
& & \langle intexp \rangle < \langle intexp \rangle \\
& & \langle intexp \rangle \leq \langle intexp \rangle \\
& & \langle intexp \rangle > \langle intexp \rangle \\
& & \langle intexp \rangle \geq \langle intexp \rangle \\
& & \neg \langle assert \rangle \\
& & \langle assert \rangle \vee \langle assert \rangle \\
& & \langle assert \rangle \wedge \langle assert \rangle \\
& & \exists \langle var \rangle . \langle assert \rangle \\
& & \forall \langle var \rangle . \langle assert \rangle
\end{array}
$$

# Particularidades de la sintaxis

- Disponemos ahora de 3 tipos de frases, las cuales por supuesto tendrán 3 dominios distintos de significado.
- $\langle var \rangle$ carece de producciones. No especificaremos qué es el conjunto de frases $\langle var \rangle$, pero asumimos que es un conjunto infinito de símbolos. Usualmente utilizaremos las metavariables $u, v, w, z$ para referirnos a elementos de $\langle var \rangle$. (Este es otro aspecto que contribuye a la condición de ”abstracta“ de la gramática. )
- Aparecen construcciones (para todo, existe) que poseen subfrases de distinto tipo.

## Convenciones para la representación de la estructura abstracta de las frases:

Utilizaremos las convenciones usuales de precedencia de operadores aritméticos, sumadas a la convención de que el alcance de un cuantificador $\forall$ o $\exists$ se extiende hasta el final de la frase, o hasta la aparición de un paréntesis que cierra cuyo alcance contiene al cuantificador.

# Estados y Función semántica

$\Sigma$ será el conjunto de todos los estados posibles. Definimos el *conjunto de estados posibles* como la familia de todas las funciones totales de $\langle var \rangle$ en $\mathbf{Z}$:

$$\Sigma = \langle var \rangle \to \mathbf{Z}$$

## Función Semántica

$$[\![ \_ ]\!]^{intexp} : \langle intexp \rangle \to \Sigma \to \mathbf{Z}$$

$$[\![ e ]\!]^{intexp} : \Sigma \to \mathbf{Z} \quad \text{(para cualquier expresión } e\text{)}$$

$$[\![ e ]\!]^{intexp}_{\sigma} \in \mathbf{Z} \quad \text{(para cualquier expresión } e\text{)}$$

# Operaciones auxiliares sobre los estados

Sea $\sigma \in \Sigma$ un estado, $v$ una variable y $n$ un entero.

$[\sigma|v : n]$ será el estado que coincide con $\sigma$ en todas las variables salvo posiblemente en $v$, donde este nuevo estado tiene asignado $n$.

$$[\sigma|v : n]w \quad = \quad \text{si } (w = v) \text{ entonces } n \text{ sino } \sigma w$$

# Función semántica de las exp. booleanas

$$[\![\_]\!]^{assert} : \langle assert \rangle \to \Sigma \to \{V, F\}$$

$$[\![p]\!]^{assert} : \Sigma \to \{V, F\} \qquad \text{(para cualquier predicado } p\text{)}$$

Notar que son dos funciones, una para expresiones enteras $[\![\_]\!]^{intexp}$ y otra para predicados $[\![\_]\!]^{assert}$, pero para no recargar la notación usaremos indistintamente la notación $[\![\_]\!]$ para ambas.

# Ecuaciones semánticas de las exp. booleanas

$$
\begin{array}{lcl}
[\![\textbf{true}]\!]\sigma & = & V \\
[\![\textbf{false}]\!]\sigma & = & F \\
[\![e = e']\!]\sigma & = & ([\![e]\!]\sigma = [\![e']\!]\sigma) \\
 & \vdots & \\
[\![p \wedge q]\!]\sigma & = & ([\![p]\!]\sigma \wedge [\![q]\!]\sigma) \\
 & \vdots & \\
[\![\forall v.p]\!]\sigma & = & (\forall n \in \mathbf{Z}.\ [\![p]\!][\sigma | v : n]) \\
[\![\exists v.p]\!]\sigma & = & (\exists n \in \mathbf{Z}.\ [\![p]\!][\sigma | v : n])
\end{array}
$$

# ¿Cuál es el significado de 1/0?

Acá debemos cuidarnos de no trasladar al lenguaje los vicios del metalenguaje, por ejemplo los problemas relacionados con la división por 0.
En esta primera aproximación a la semántica denotacional vamos a asumir que todas las funciones son totales. En particular, la división está siempre definida (como función del dominio semántico), incluso si el divisor es 0.
Más adelante abordaremos la manera en que los lenguajes gestionan situaciones de error o no definición.

# Variables y metavariables

**Noción de metavariable**: es una variable del metalenguaje.

**Ejemplos** $e$ y $e'$ (corren sobre expresiones enteras), $p$ y $q$ (corren sobre los predicados), $n$ (corre sobre enteros) y $\sigma$ (corre sobre estados), $v, u, w$ (corre sobre las variables del lenguaje)

Usaremos $u, v, w$ para metavariables y x, y, z para variables del lenguaje

# Ligadura

- **Ocurrencia ligadora**: una ocurrencia ligadora de una variable es la que se encuentra inmediatamente después de un cuantificador ($\forall$ o $\exists$).
- **Alcance de una ocurrencia ligadora**: En $\forall v.\ p$ o $\exists v.\ p$, el predicado $p$ es el alcance de la ocurrencia ligadora de $v$.
- **Ocurrencia ligada**: cualquier ocurrencia de $v$ en el alcance de una ocurrencia ligadora de $v$ es una ocurrencia ligada de $v$ (se toma la de menor alcance).
- **Ocurrencia libre**: una ocurrencia de una variable que no es ligadora ni ligada es una ocurrencia libre.
- **Variable libre**: una variable que tiene ocurrencias libres es una variable libre.
- **Expresión cerrada**: no tiene variables libres.

# Variables libres

$$
\begin{array}{lcl}
FV\ \lfloor n \rfloor & = & \emptyset \\
FV\ v & = & \{v\} \\
FV(-e) & = & FV\ e \\
FV(e + e') & = & FV\ e \cup FV\ e' \\
 & \vdots &
\end{array}
\qquad
\begin{array}{lcl}
FV\ \textbf{true} & = & \emptyset \\
FV\ \textbf{false} & = & \emptyset \\
FV(e = e') & = & FV\ e \cup FV\ e' \\
 & \vdots & \\
FV(\neg p) & = & FV\ p \\
FV(p \wedge q) & = & FV\ p \cup FV\ q \\
 & \vdots & \\
FV(\forall v.\ p) & = & (FV\ p) - \{v\} \\
FV(\exists v.\ p) & = & (FV\ p) - \{v\}
\end{array}
$$

# Operador Sustitución

Las expresiones con variables libres pueden instanciarse sustituyendo sus variables libres por términos.

**Conjunto de sustituciones:**

$$\Delta = \langle var \rangle \rightarrow \langle intexp \rangle$$

El operador sustitución "opera" sobre expresiones enteras (términos) y expresiones booleanas (predicados).

$$\_/\_ \in \langle intexp \rangle \times \Delta \rightarrow \langle intexp \rangle$$

$$\_/\_ \in \langle assert \rangle \times \Delta \rightarrow \langle assert \rangle$$

# Operador Sustitución

$$
\begin{aligned}
0/\delta &= 0... \\
v/\delta &= \delta v \\
(-e)/\delta &= -(e/\delta) \\
(e+f)/\delta &= (e/\delta)+(f/\delta)...
\end{aligned}
$$

$$
\begin{aligned}
(e=f)/\delta &= (e/\delta)=(f/\delta) \\
(\neg p)/\delta &= \neg(p/\delta)... \\
(p \vee q)/\delta &= (p/\delta) \vee (q/\delta)...
\end{aligned}
$$

# El problema de la captura

**Aplicación "ingenua" de la sustitución**:

si $e = \delta$y entonces

$$(\exists \text{x. x} > \text{y})/\delta \quad = \quad \exists \text{x. x} > e$$

Pero si $e$ es x, quedaría $\exists$x. x $>$ x que es falsa, a diferencia de $\exists$x. x $>$ y que es verdadera.

# El problema de la captura

**Visto de otra manera**:

$\exists$x. x $>$ y
$\exists$z. z $>$ y

sólo se diferencian en el nombre de la variable ligada, de manera que al sustituir en uno u otro me debería dar resultados equivalentes. Pero para $\delta$y $=$ x $+$ 1 obtenemos predicados no equivalentes.

**Problema:** se está "capturando" x, que era libre, y ahora pasa a ser ligada.

# Solución al problema de la captura

Renombramos la variable ligadora y ligada antes de efectuar una susutitución que producirá una captura.

Renombramos eligiendo una variable "nueva", que llamaremos $v_{new}$. Esto es, una variable no "capturable". Las variables "capturables" son las que pueden aparecer al aplicar la sustitución.

$$\begin{aligned}(\forall v \,.\, b)/\delta &= \forall v_{new} \,.\, (b/[\delta|v : v_{new}]) \\ (\exists v \,.\, b)/\delta &= \exists v_{new} \,.\, (b/[\delta|v : v_{new}])\end{aligned}$$

donde

$$v_{new} \notin \bigcup_{w \in FV(b)-\{v\}} FV(\delta\ w)$$

# Propiedades de la semántica

Hay dos propiedades de la semántica que resultan relevantes para los lenguajes de programación.

**Teorema de Coincidencia (TC)**
Si dos estados $\sigma$ y $\sigma'$ coinciden en las variables libres de $p$, entonces da lo mismo evaluar $p$ en $\sigma$ o $\sigma'$. En símbolos:

$$(\forall w \in FV(p) \,.\, \sigma\ w = \sigma'\ w) \Longrightarrow [\![p]\!]\sigma = [\![p]\!]\sigma'.$$

**Teorema de Renombre (TR)**
Los nombres de la variables ligadas no tienen importancia. En símbolos,

$$u \notin FV(q) - \{v\} \Rightarrow [\![\forall u \,.\, q/v \to u]\!] = [\![\forall v \,.\, q]\!]$$

# Teorema de Sustitución

Si aplico la sustitución $\delta$ a $p$ y luego evalúo en el estado $\sigma$, puedo obtener el mismo resultado a partir de $p$ sin sustituir, si evalúo en un estado que hace el trabajo de $\delta$ y de $\sigma$ (en las variables libres de $p$).
En símbolos:

$$(\forall w \in FV(p) \,.\, [\![\delta\ w]\!]\sigma = \sigma'\ w) \Longrightarrow [\![p/\delta]\!]\sigma = [\![p]\!]\sigma'.$$

# Un lenguaje imperativo simple (sin iteración)

$$
\begin{array}{lcl}
\langle comm \rangle & ::= & \textbf{skip} \\
 & & \langle var \rangle := \langle intexp \rangle \\
 & & \langle comm \rangle \ ; \ \langle comm \rangle \\
 & & \textbf{if } \langle boolexp \rangle \textbf{ then } \langle comm \rangle \textbf{ else } \langle comm \rangle \\
 & & \textbf{newvar } \langle var \rangle := \langle intexp \rangle \textbf{ in } \langle comm \rangle
\end{array}
$$

$$
\begin{array}{lll}
\langle intexp \rangle & ::= & \langle natconst \rangle \\
 & & \langle var \rangle \\
 & & \langle intexp \rangle + \langle intexp \rangle \\
 & & \langle intexp \rangle * \langle intexp \rangle \\
 & & \langle intexp \rangle - \langle intexp \rangle \\
 & & \langle intexp \rangle \,/\, \langle intexp \rangle \\
 & & \langle intexp \rangle \,\%\, \langle intexp \rangle \\
 & & - \langle intexp \rangle
\end{array}
$$

$$
\begin{array}{lll}
\langle boolexp \rangle & ::= & \langle boolconst \rangle \\
 & & \langle intexp \rangle = \langle intexp \rangle \\
 & & \langle intexp \rangle < \langle intexp \rangle \\
 & & \langle intexp \rangle \leq \langle intexp \rangle \\
 & & \langle intexp \rangle > \langle intexp \rangle \\
 & & \langle intexp \rangle \geq \langle intexp \rangle \\
 & & \neg \langle boolexp \rangle \\
 & & \langle boolexp \rangle \vee \langle boolexp \rangle \\
 & & \langle boolexp \rangle \wedge \langle boolexp \rangle
\end{array}
$$

$$
\begin{array}{lll}
\langle natconst \rangle & ::= & 0 \mid 1 \mid 2 \mid .... \\
\langle boolconst \rangle & ::= & \textbf{true} \mid \textbf{false}
\end{array}
$$

# Características de LIS

- Hay sólo dos tipos de expresiones: enteras y booleanas

# Características de LIS

- Hay sólo dos tipos de expresiones: enteras y booleanas
- Las expresiones enteras siempre se pueden evaluar, y su resultado es un entero.Todas las funciones primitivas (incluida la división) son funciones totales.

# Características de LIS

- Hay sólo dos tipos de expresiones: enteras y booleanas
- Las expresiones enteras siempre se pueden evaluar, y su resultado es un entero.Todas las funciones primitivas (incluida la división) son funciones totales.
- Sólo posee variables que adoptan valores enteros. Luego la noción de **estado** se refleja en la siguiente definición:

$$\textbf{Conjunto de estados:} \qquad \Sigma = \langle var \rangle \rightarrow \mathbf{Z}$$

(la memoria posee infinitos lugares que siempre alojan un número entero).

# Significado de los comandos de LIS (sin iteración)

**Funciones semánticas:**

$[\![\_]\!]^{intexp} \in \langle intexp \rangle \to \Sigma \to \mathbf{Z}$

$[\![\_]\!]^{boolexp} \in \langle boolexp \rangle \to \Sigma \to \{V, F\}$

$[\![\_]\!]^{comm} \in \langle comm \rangle \to \Sigma \to \Sigma_{\perp}$

# Primeras ecuaciones semánticas

$$[\![\textbf{skip}]\!]\sigma \quad = \quad \sigma$$

# Primeras ecuaciones semánticas

$$
\begin{aligned}
[\![\textbf{skip}]\!]\sigma &= \sigma \\
[\![v := e]\!]\sigma &= [\sigma | v : [\![e]\!]\sigma]
\end{aligned}
$$

# Primeras ecuaciones semánticas

$$
\begin{array}{lcl}
[\![\textbf{skip}]\!]\sigma & = & \sigma \\
[\![v := e]\!]\sigma & = & [\sigma | v : [\![e]\!]\sigma] \\
[\![\textbf{if } e \textbf{ then } c \textbf{ else } c']\!]\sigma & = & \text{if } [\![e]\!]\sigma \text{ then } [\![c]\!]\sigma \text{ else } [\![c']\!]\sigma
\end{array}
$$

# Primeras ecuaciones semánticas

$$
\begin{array}{lcl}
[\![\textbf{skip}]\!]\sigma & = & \sigma \\
[\![v := e]\!]\sigma & = & [\sigma | v : [\![e]\!]\sigma] \\
[\![\textbf{if } e \textbf{ then } c \textbf{ else } c']\!]\sigma & = & \text{if } [\![e]\!]\sigma \text{ then } [\![c]\!]\sigma \text{ else } [\![c']\!]\sigma
\end{array}
$$

**Ejemplo:**

$$[\![x := x - 1]\!]_\sigma \quad = \quad [\sigma | x : [\![x - 1]\!]_\sigma]$$

# Primeras ecuaciones semánticas

$$
\begin{array}{lcl}
[\![\textbf{skip}]\!]\sigma & = & \sigma \\
[\![v := e]\!]\sigma & = & [\sigma | v : [\![e]\!]\sigma] \\
[\![\textbf{if } e \textbf{ then } c \textbf{ else } c']\!]\sigma & = & \text{if } [\![e]\!]\sigma \text{ then } [\![c]\!]\sigma \text{ else } [\![c']\!]\sigma
\end{array}
$$

**Ejemplo:**

$$
\begin{array}{lcl}
[\![x := x - 1]\!]_\sigma & = & [\sigma | x : [\![x - 1]\!]_\sigma] \\
 & = & [\sigma | x : [\![x]\!]_\sigma - [\![1]\!]_\sigma]
\end{array}
$$

# Primeras ecuaciones semánticas

$$
\begin{aligned}
[\![\textbf{skip}]\!]\sigma &= \sigma \\
[\![v := e]\!]\sigma &= [\sigma | v : [\![e]\!]\sigma] \\
[\![\textbf{if } e \textbf{ then } c \textbf{ else } c']\!]\sigma &= \text{if } [\![e]\!]\sigma \text{ then } [\![c]\!]\sigma \text{ else } [\![c']\!]\sigma
\end{aligned}
$$

**Ejemplo:**

$$
\begin{aligned}
[\![x := x - 1]\!]_\sigma &= [\sigma | x : [\![x - 1]\!]_\sigma] \\
&= [\sigma | x : [\![x]\!]_\sigma - [\![1]\!]_\sigma] \\
&= [\sigma | x : \sigma\ x - 1]
\end{aligned}
$$

# Semántica de la composición de comandos

La ecuación $[\![c_0; c_1]\!]\sigma = [\![c_1]\!]([\![c_0]\!]\sigma)$ no está bien tipada.

Esto se soluciona acudiendo a funciones auxiliares de transferencia de control.

# Semántica de la composición de comandos

La ecuación $[\![c_0; c_1]\!]\sigma = [\![c_1]\!]([\![c_0]\!]\sigma)$ no está bien tipada.

Esto se soluciona acudiendo a funciones auxiliares de transferencia de control.

Si $f \in \Sigma \to \Sigma_\perp$, entonces definimos una nueva función $f_{\perp\!\!\perp} \in \Sigma_\perp \to \Sigma_\perp$ de la siguiente manera:

# Semántica de la composición de comandos

La ecuación $[\![c_0; c_1]\!]\sigma = [\![c_1]\!]([\![c_0]\!]\sigma)$ no está bien tipada.

Esto se soluciona acudiendo a funciones auxiliares de transferencia de control.

Si $f \in \Sigma \to \Sigma_\perp$, entonces definimos una nueva función $f_{\perp\!\!\perp} \in \Sigma_\perp \to \Sigma_\perp$ de la siguiente manera:

$$f_{\perp\!\!\perp}\sigma = f\ \sigma \qquad\qquad f_{\perp\!\!\perp}\ \perp = \perp$$

# Semántica de la composición de comandos

La ecuación $[\![c_0; c_1]\!]\sigma = [\![c_1]\!]([\![c_0]\!]\sigma)$ no está bien tipada.

Esto se soluciona acudiendo a funciones auxiliares de transferencia de control.

Si $f \in \Sigma \to \Sigma_\perp$, entonces definimos una nueva función $f_{\perp\!\!\perp} \in \Sigma_\perp \to \Sigma_\perp$ de la siguiente manera:

$$f_{\perp\!\!\perp}\sigma = f\ \sigma \qquad\qquad f_{\perp\!\!\perp}\ \perp = \perp$$

Ecuación semántica para la composición:

$$[\![c_0; c_1]\!]\sigma = [\![c_1]\!]_{\perp\!\!\perp}([\![c_0]\!]\sigma)$$

# Semántica de **newvar**

Función “restauración de $v$ según $\sigma$”:

$$f_{v,\sigma}\sigma' = [\sigma'|v : \sigma v]$$

# Semántica de **newvar**

Función "restauración de $v$ según $\sigma$":

$$f_{v,\sigma}\sigma' = [\sigma'|v : \sigma v]$$

$$[\![\textbf{newvar } v := e \textbf{ in } c]\!]\sigma \quad = \quad (f_{v,\sigma})_{\perp\!\perp}([\![c]\!][\sigma|v : [\![e]\!]\sigma])$$

# Semántica de **newvar**

Función "restauración de $v$ según $\sigma$":

$$f_{v,\sigma}\sigma' = [\sigma'|v : \sigma v]$$

$$[\![\textbf{newvar } v := e \textbf{ in } c]\!]\sigma \quad = \quad (f_{v,\sigma})_{\perp\!\!\perp}([\![c]\!][\sigma|v : [\![e]\!]\sigma])$$

Notación lambda para la función restauración:

$$f_{v,\sigma} \quad = \quad \lambda\sigma' \in \Sigma.\ [\sigma'|v : \sigma v]$$

$$[\![\textbf{newvar } v := e \textbf{ in } c]\!]\sigma \quad = \quad (\lambda\sigma' \in \Sigma.\ [\sigma'|v : \sigma v])_{\perp\!\!\perp}\ ([\![c]\!][\sigma|v : [\![e]\!]\sigma])$$

# Un lenguaje imperativo simple (sin iteración)

$$
\begin{array}{lcl}
\langle comm \rangle & ::= & \textbf{skip} \\
 & & \langle var \rangle := \langle intexp \rangle \\
 & & \langle comm \rangle \ ; \ \langle comm \rangle \\
 & & \textbf{if } \langle boolexp \rangle \textbf{ then } \langle comm \rangle \textbf{ else } \langle comm \rangle \\
 & & \textbf{newvar } \langle var \rangle := \langle intexp \rangle \textbf{ in } \langle comm \rangle
\end{array}
$$

$$
\begin{array}{lll}
\langle intexp \rangle & ::= & \langle natconst \rangle \\
 & & \langle var \rangle \\
 & & \langle intexp \rangle + \langle intexp \rangle \\
 & & \langle intexp \rangle * \langle intexp \rangle \\
 & & \langle intexp \rangle - \langle intexp \rangle \\
 & & \langle intexp \rangle \,/\, \langle intexp \rangle \\
 & & \langle intexp \rangle \,\%\, \langle intexp \rangle \\
 & & - \langle intexp \rangle
\end{array}
$$

$$
\begin{array}{lll}
\langle boolexp \rangle & ::= & \langle boolconst \rangle \\
 & & \langle intexp \rangle = \langle intexp \rangle \\
 & & \langle intexp \rangle < \langle intexp \rangle \\
 & & \langle intexp \rangle \leq \langle intexp \rangle \\
 & & \langle intexp \rangle > \langle intexp \rangle \\
 & & \langle intexp \rangle \geq \langle intexp \rangle \\
 & & \neg \langle boolexp \rangle \\
 & & \langle boolexp \rangle \vee \langle boolexp \rangle \\
 & & \langle boolexp \rangle \wedge \langle boolexp \rangle
\end{array}
$$

$$
\begin{array}{lll}
\langle natconst \rangle & ::= & 0 \mid 1 \mid 2 \mid .... \\
\langle boolconst \rangle & ::= & \textbf{true} \mid \textbf{false}
\end{array}
$$

# Características de LIS

- Hay sólo dos tipos de expresiones: enteras y booleanas
- Las expresiones enteras siempre se pueden evaluar, y su resultado es un entero.Todas las funciones primitivas (incluida la división) son funciones totales.
- Sólo posee variables que adoptan valores enteros. Luego la noción de **estado** se refleja en la siguiente definición:

$$\textbf{Conjunto de estados:} \qquad \Sigma = \langle var \rangle \rightarrow \mathbf{Z}$$

(la memoria posee infinitos lugares que siempre alojan un número entero).

# Significado de los comandos de LIS (sin iteración)

**Funciones semánticas:**

$[\![\_]\!]^{intexp} \in \langle intexp \rangle \to \Sigma \to \mathbf{Z}$

$[\![\_]\!]^{boolexp} \in \langle boolexp \rangle \to \Sigma \to \{V, F\}$

$[\![\_]\!]^{comm} \in \langle comm \rangle \to \Sigma \to \Sigma_{\perp}$

# Primeras ecuaciones semánticas

$$
\begin{aligned}
[\![\textbf{skip}]\!]\sigma &= \sigma \\
[\![v := e]\!]\sigma &= [\sigma | v : [\![e]\!]\sigma] \\
[\![\textbf{if } e \textbf{ then } c \textbf{ else } c']\!]\sigma &= \text{if } [\![e]\!]\sigma \text{ then } [\![c]\!]\sigma \text{ else } [\![c']\!]\sigma \\
[\![c_0; c_1]\!]\sigma &= [\![c_1]\!]([\![c_0]\!]\sigma)
\end{aligned}
$$

# Ejemplo:

$$
\begin{aligned}
[\![x := x - 1;\ x := 0]\!]\sigma &= [\![x := 0]\!]([\![x := x - 1]\!]\sigma) \\
&= [\![x := 0]\!]([\sigma | x : [\![x - 1]\!]\sigma]) \\
&= [\![x := 0]\!]([\sigma | x : [\![x]\!]\sigma - [\![1]\!]\sigma]) \\
&= [\![x := 0]\!]([\sigma | x : \sigma\ x - 1]) \\
&= [[\sigma | x : \sigma\ x - 1] | x : 0] \\
&= [\sigma | x : 0]
\end{aligned}
$$

# Semántica de **newvar**

Función "restauración de $v$ según $\sigma$":

$$f_{v,\sigma}\sigma' = [\sigma'|v : \sigma v]$$

$$[\![\textbf{newvar } v := e \textbf{ in } c]\!]\sigma \quad = \quad (f_{v,\sigma})([\![c]\!][\sigma|v : [\![e]\!]\sigma])$$

Notación lambda para la función restauración:

$$f_{v,\sigma} \quad = \quad \lambda\sigma' \in \Sigma.\ [\sigma'|v : \sigma v]$$

$$[\![\textbf{newvar } v := e \textbf{ in } c]\!]\sigma \quad = \quad (\lambda\sigma' \in \Sigma.\ [\sigma'|v : \sigma v])\ ([\![c]\!][\sigma|v : [\![e]\!]\sigma])$$

# Dificultades para dar significado a la iteración

El comando

$$\textbf{while } b \textbf{ do } c$$

debe satisfacer la propiedad:

$$[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \text{if } [\![b]\!]\sigma \text{ then } [\![c; \textbf{while } b \textbf{ do } c]\!]\sigma \text{ else } [\![\textbf{skip}]\!]\sigma$$

$$= \begin{cases} [\![\textbf{while } b \textbf{ do } c]\!]([\![c]\!]\sigma) & \text{si } [\![b]\!]\sigma \\ \sigma & \text{si } \neg[\![b]\!]\sigma \end{cases}$$

# Dificultades para dar significado a la iteración

Dificultad principal:

$$[\![\textbf{while } b \textbf{ do } c]\!]\sigma \quad = \quad \begin{cases} [\![\textbf{while } b \textbf{ do } c]\!]([\![c]\!]\sigma) & \text{si } [\![b]\!]\sigma \\ \sigma & \text{si } \neg[\![b]\!]\sigma \end{cases} \qquad (W)$$

no es dirigida por sintaxis

# Dificultades para dar significado a la iteración

La propiedad del **while** no puede ser tomada como definición.

Sea $\omega = [\![\textbf{while } b \textbf{ do } c]\!]$. Debe notarse la similitud entre la ecuación

$$\omega\sigma = \begin{cases} \omega([\![c]\!]\sigma) & \text{si } [\![b]\!]\sigma \\ \sigma & \text{si } \neg[\![b]\!]\sigma \end{cases} \qquad (W)$$

y las definiciones usuales de funciones utilizando recursión, como por ejemplo

$$f\ n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f(n-2) & \text{caso contrario (c.c.)} \end{cases} \qquad (ER)$$