Análisis Comparativo de Lenguajes (cod. 1956)

Ciencias de la Computación

Dpto. de Computación - FCEFQyN - UNRC

Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto

2016

Presentación de la Asignatura

Teóricos

Ariel Gonzalez.
Miércoles de 11 a 14hs., Aula 18 Pab.4I

Prácticos

Comisión 1:
Lunes de 9 a 12 hs., Lab. 102 Pab. 2 y Viernes de 10 a 12 hs., Lab. 102 Pab. 2.
Comisión 2:
Martes de 16 a 19hs, Lab. 102 Pab. 2 y Jueves de 14 a 16 hs., Lab. 101 Pab. 2.

Docentes de los prácticos

Prof. Maria M. Novaira - Lic. Valeria Bengolea - Mg. Ariel Gonzalez - Prof. Angeli Sandra
Ay. de Segunda: a confirmar

Presentación de la Asignatura

Respecto a las Consultas

- La secretaría del departamento no brindará información a cerca de los contenidos y horarios de la materia.
- No se responderán consultas telefónicas en ningúm momento.
- No se reponderán consultas fuera de los horarios establecidos.
- Toda información referida a la materia, se encontrará exclusivamente en el sitio web de la asignatura.

Presentación de la Asignatura

Modalidad y Calendario

El régimen de regularización de la materia, exige la aprobación de 2 exámenes parciales con sus respectivos recuperatorios.

- Primer Parcial: Viernes 30-09
- Primer Recuperatorio: Jueves 06-10.
- Segundo Parcial: Viernes 10-11.
- Segundo Recuperatorio: Jueves 17-11.
- Los exámenes deberán aprobarse con nota mínima: 5 (cinco).

Temario

- Estudio de los Conceptos y Principios usados en el diseño de lenguajes de programación
- Análisis de características y comparación de lenguajes
- Paradigmas de programación (imperativo, orientada a objetos, funcional y logica)
- Concurrencia
- Semántica de lenguajes de programación

Objetivos

- Mayor capacidad de análisis y comparación de lenguajes
- Conocimiento de detalles de implementación
- Mejor utilización de los lenguajes
- Mejor utilización de herramientas de desarrollo

Bibliografía y herramientas a utilizar

Bibliografía

- *Programming Language Concepts and Paradigms*. David Watt
- *Programming Languages: Design and Implementation. 2nd Ed.* Terrence Pratt, Marvin Zelkowitz
- *Concepts, Techniques and Models of Computer Programming*. Peter Van Roy and Seif Haridi
- Tutoriales y manuales de los lenguajes a utilizar

Lenguajes de programación a utilizar

- Imperativos: Pascal, C, Python, etc.
- Funcionales: Lisp, Haskell, etc.
- Orientados a objetos: C++, Java, Python, etc.
- Lógicos: Prolog
- Multiparadigma: Oz
- Concurrencia: Java y Erlang

Un poco de historia

Años	Evolución
1951-55	Assembly, Lenguajes de expresiones (A0)
1956-60	Fortran, Algol 58, COBOL, Lisp
1961-65	Snobol, Algol 60, COBOL 61, Jovial, APL
1966-70	Fortran 66, COBOL 65, Algol 68, Simula, BASIC, PL/I
1971-75	Pascal, COBOL 74, C, Scheme, Prolog
1976-80	Smaltalk, Ada, Fortran 77, ML
1981-85	Turbo Pascal, Smaltalk 80, Ada 83, Postcript
1986-90	Fortran 90, C++, SML, Haskell
1991-95	Ada 95, TCL, Perl, Java, Ruby, Python
1996-00	Ocaml, Delphy, Eiffel, JavaScript
2000-	D, C#, Fortran 2003, Starlog, TOM, . . .

Diseño de lenguajes de programación

Qué es la programación?

Diseñar y utilizar (componer) abstracciones para obtener un objetivo.

Objetivo principal

Permitir definir ***abstracciones*** para describir *procesos computacionales* en forma **modular**.

- *Abstracción procedural*: abstrae *computaciones*.
- *Abstracción de datos*: abstrae la representación (implementación) de los datos.

Lenguajes de Programación - Propiedades requeridas

- **Universal**: cada problema computable debería tener una solución programable en el lenguaje.
- **Natural:** con respecto a su dominio de aplicación. Por ejemplo, un lenguaje orientado a problemas numéricos debería ser muy rico en tipos numéricos, vectores y matrices.
- **Implementable**: debería ser posible escribir un interprete o compilador en algún sistema de computación.
- **Eficiente: en recursos**
- **Simple**
- **Uniforme**
- **Legible**
- **Seguro**

Sintaxis y semántica

- **Sintaxis**: tiene que ver con la *forma* que adoptan los programas.
- **Semántica**: tiene que ver con el *significado* de los programas, es decir de su *comportamiento* cuando son ejecutados.

Definiciones formales de sintaxis

Gramáticas

Gramáticas: conjunto de *reglas* o *producciones* que especifican cadenas válidas de palabras del lenguaje

- *regulares*: las reglas son de la forma
  $NonTerminal \rightarrow Terminal\ NonTerminal \mid Terminal$
  Equivalentes a las *expresiones regulares*
- *libres de contexto y BNFs*: permiten definir frases estructuradas. Las reglas son de la forma:
  $NonTerminal \rightarrow (NonTerminal \mid Terminal)^*$

Autómatas

Formalismos de aceptación de cadenas de un lenguaje.

- *finitos*: aceptan lenguajes regulares
- *pila*: aceptan lenguajes libres de contexto

Gramáticas libres de contexto

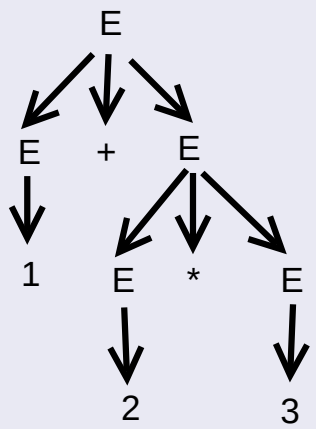
Gramática libre de contexto

$$G =< N, T, S, P >$$

- N: conjunto de símbolos ***no terminales (variables)***
- T: conjunto de símbolos***terminales***
- P: conjunto de producciones de la forma $\subseteq N \times (N \times T)^*$
- S: símbolo de comienzo.

Ejemplo de una gramática libre de contexto (expresiones aritméticas):
E -> E + E | E - E | E * E | E / E | (E) | D
D -> 0 | 1 | . . . | 9

Árbol sintáctico o de derivación



Un árbol de derivación para la cadena *1+2*3*
La gramática es *ambigua*.

Extended Backus-Naur Form (EBNF)

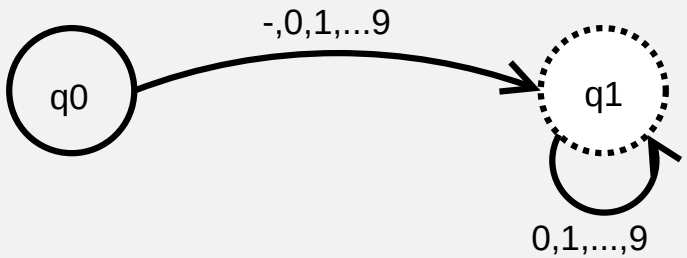
- Provee una forma más cómoda de describir gramáticas libres de contexto.
- La parte derecha de una producción es reemplazada por una expresión regular:
  - Parte opcional:
    $N \rightarrow \ldots [\text{opcional}] \ldots$
  - Repeticiones:
    $N \rightarrow \ldots (0 \text{ o más veces}) * \ldots$
    $N \rightarrow \ldots (1 \text{ o más veces}) + \ldots$
- Permite la implementación inmediata de reconocedores (parsers):
  1. Cada *no terminal* N es un procedimiento.
  2. El cuerpo del procedimiento reconoce *tokens* (terminales) e invoca a otros procedimientos.

Reconocedores de lenguajes (parsers)

- El *parser* implementa un reconocedor (ej:autómata pila) para una gramática libre de contexto.
- El parser se apoya en el *scanner* para reconocer los símbolos terminales (*tokens*).
- El scanner reconoce un lenguaje regular, por lo que implementa un autómata finito.
- A las características *dependientes del contexto* (ej: uso de un identificador dentro de su alcance) las resuelve el *analizador semántico*.
- Existen herramientas para generar código fuente de reconocedores de lenguajes a partir de la especificación de la gramática. Ej: *lex* y *yacc*.

Definiciones formales de sintaxis (cont.)

- Expresión regular que acepta números enteros: $(-)?[0-9]+$
- Autómata finito que acepta un lenguaje regular equivalente:



(el estado q_1 es un estado final)
Ejercicio: definir una gramática regular equivalente.

Lenguajes de Programación

Sintaxis

Definición de las frases *legales* del lenguaje.
Especificada por una gramática o EBNF.

Semántica

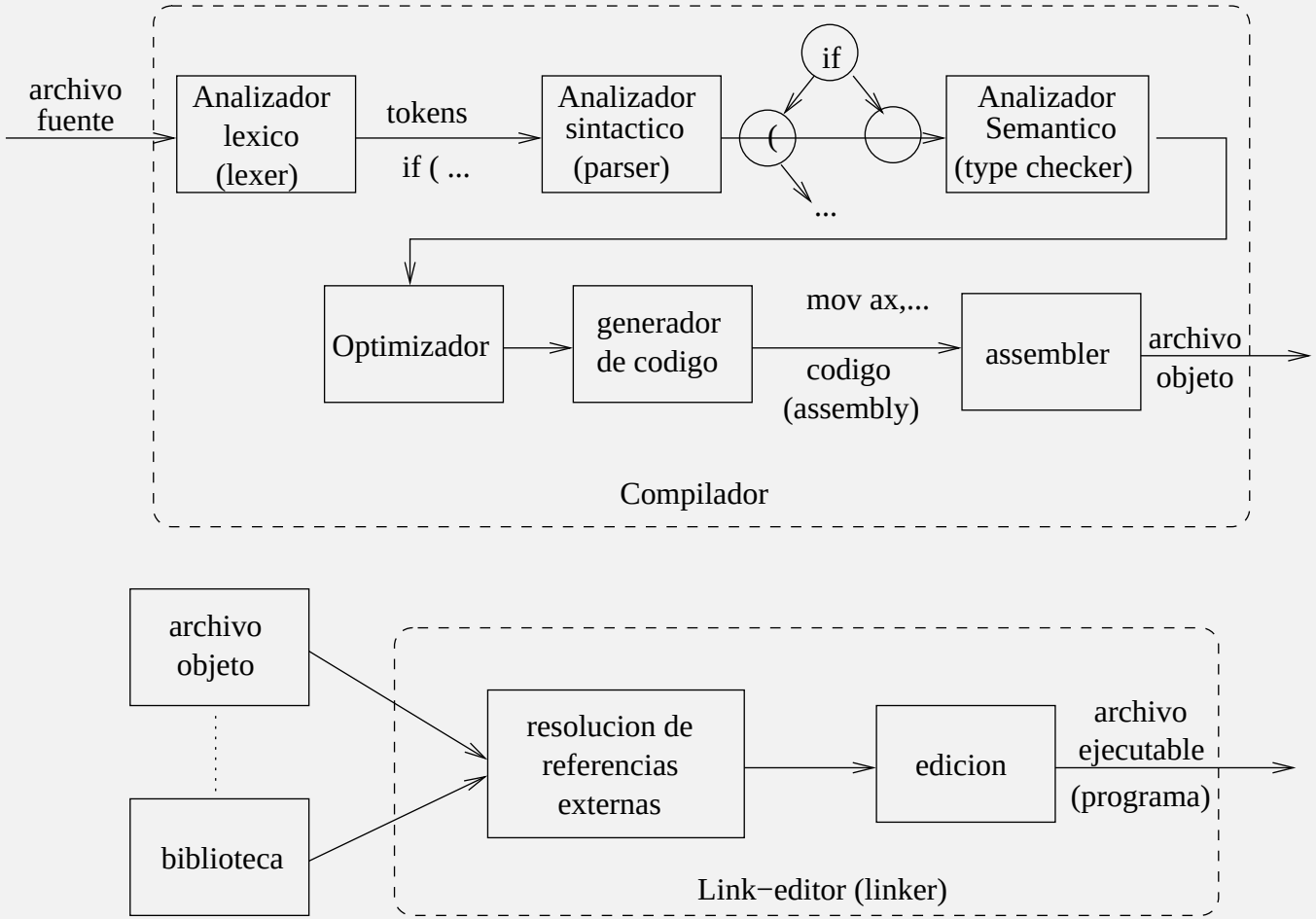
Significado de las frases del lenguaje.

- **Operacional**: cómo se ejecutan las sentencias en una máquina abstracta.
- **Denotacional**: define una sentencia como una función sobre un dominio abstracto.
- **Axiomática**: se definen sentencias como una relación entre estados de entrada y salida.
- **Lógica**: se definen las sentencias como un modelo de una teoría lógica.

Procesadores de lenguajes de programación

- **Compilador**: traduce un programa *fuente* a código assembly u objeto (archivo binario enlazable)
- **Linker o enlazador** combina archivos objetos y bibliotecas, resolviendo referencias externas y genera un *programa* (archivo binario ejecutable)
- **Intérprete**: a partir del código fuente genera una *representación interna* la cual es *evaluada o ejecutada* por el mismo intérprete(Basic,Haskel,ML,SmallTalk, etc)
- **Ejecutor o máquinas virtuales**: es posible generar código de una máquina vitual (no hardware real), ej: JVM. El código es ejecutado por medio de un *ejecutor o intérprete* (ej: COBOL, Java)

El proceso de compilación



Archivos binarios

Archivos objeto

Generalmente contienen diferentes tipos de datos:

- **Block Started by Symbol (BSS)**: área de datos estática inicializada en cero.
- **Text (code) segment**: contiene instrucciones de máquina. Generalmente de tamaño fijo y sólo lectura.
- **Data segment**: área estática que contiene los valores de las variables globales inicializadas en el programa.
- **Tabla de símbolos**: mapping de identificadores a sus direcciones de memoria (o en blanco en caso que sean referencias externas, es decir símbolos definidos en otros módulos).

Archivos binarios (cont.)

Archivos bibliotecas (libraries)

Colección de subprogramas (funciones, procedimientos, datos) relacionados.
No tienen una dirección de comienzo.

- **Estáticas**: componentes de la biblioteca son *insertados* en cada programa que la utilize. Ventajas y Desventajas.
- **Dinámicas**: los componentes se cargan durante la ejecución de un programa. Ventajas y Desventajas.

Archivos ejecutables (programas)

Son el resultado del enlazado de los diferentes archivos objetos y bibliotecas que componen un programa.
Generalmente contiene segmentos como los archivos objeto.

Capítulo 2

Lenguajes y modelos de programación

Características de los Modelos o Paradigmas de programación.

Modelos de computación

Programación

- Un *modelo de computación*: lenguaje y semántica de ejecución.
- *Técnicas de programación*.
- *Técnicas de **razonamiento** sobre programas.*

Modelos (o paradigmas)

- **Declarativo**: sin estado (memoryless)
  - Programación funcional
  - Programación lógica
- **Imperativo**: con estado (las computaciones dependen de su estado interno además de sus parámetros)
  - Componentes
  - Programación orientada a objetos
- **Concurrencia**: procesos como actividades independ.

Modelo declarativo

Ventajas

- Claridad y Simplicidad.
- Razonamiento Modular.

Desventajas

- Dificultad para modelar ciertos problemas, ej. operaciones de entrada-salida.
- Rendimientos menores respecto a programaas equivalentes con estado.

Transparencia Referencial (definición)

Analizar el ejemplo:
x := 1; $\{x = 1\}$ y := f(x) + x; $\{x = 1 \wedge y = 2 + 1\}$

Modelo con Estados

Ventajas

- Facilidad para solucionar ciertos problemas.
- Eficiencia.

Desventajas

- Perdida de razonamiento al estilo ecuacional.
- Perdida de razonamiento Modular.

Lenguajes

Lenguajes Declarativos

- *Funcionales*: Haskell, LISP, SCHEME, ML y sus derivados (Ocaml, etc.).
- *Relacionales*: Prolog y sus derivados (micro-PROLOG)
- *Memoria de Asignación única*: subconjunto de OZ.

Lenguajes con estado

- *Procedurales*: Pascal, C, Fortran, etc.
- Orientados a Objetos: JAVA, C++, etc.

Clasificación de los Elementos de un Lenguaje de Programación

- *Valores*: literales y agregados.
- *Declaraciones*: de variables, constantes, de Tipos, ...?
- *Comandos*: básicos, de secuencia o bloques, Saltos, de Selección o Condicionales, Iteración
- *Operadores*.
- *Expresiones*.

Tipos de Datos

Definición

*Un **tipo de datos** es una descripción de un conjunto de valores junto con las operaciones que se aplican sobre ellos.*

Cada valor corresponde a un *tipo de datos*.

Clasificación

- *elementales*: los números y literales.
- *estructurados*:

  Por el tipo de sus elementos que contienen:
  - **homogeneos**
  - **heterogeneos**

  Por sus características de manejo:
  - **estáticos**
  - **dinámicos**
  - **semi-dinámicos**

Chequeo de Tipos

Clasificación y Características

- Tipado estático. El chequeo se realiza en tiempo de compilación.
- Tipado dinámico. El chequeo se realiza en tiempo de ejecución.

Sistemas de tipos

- *sistema de tipos fuertes*: impide que se ejecute una operación sobre argumentos de tipo no esperado.
- *sistema de tipos débil*: realiza conversiones de tipos (*cast*). Ejemplo.
- *Polimorfismo (muchas formas)*:operaciones aceptan argumentos de una familia de tipos.
- *Tipos dependientes*: ej. los arreglos dependen de un tipo base.
- *Sistema de tipos Seguro*: un lenguaje fuertemente tipado se dice que tiene un sistema de tipos Seguro. No permite operaciones que producirían condiciones inválidas.

Declaraciones, ligadura y ambientes

Una declaración relaciona (liga) un identificador con una o mas entidades. Ej. **const Pi=3.141517**;

- Ligadura estática. Durante la compilación.
- Ligadura dinámica. Durante la ejecución.

Una declaración de alguna entidad *alcanza* una cierta porción en un programa (bloques).
Un *ambiente* (o *espacio de nombres*) es un cjto. de ligaduras.

Alcanzabilidad

- *Alcance estático*. Las sentencias de un bloque se ejecutan en el ambiente de su declaración.
- *Alcance dinámico*. Las sentencias de un bloque se ejecutan en el ambiente de su ejecución. No permite analizar un bloque de forma modular.

Equivalencia de Tipos

- Equivalencia Estructural. Requiere que dos tipos tengan la misma estructura.
- Equivalencia por nombre. Las dos expresiones de tipo deben tener el mismo nombre.

Excepciones

Definición

Es un evento que ocurre durante la ejecución como producto de alguna operación inválida, y requiere la ejecución de código fuera del flujo normal de control.

Un lenguaje debe proveer mecanismos para el manejo de excepciones (*exception handling*).

Capítulo 3

El Modelo Declarativo

- Definición de un lenguaje de programación (OZ).
- Sintaxis y Semántica Operacional.

Definición de un lenguaje de programación

El enfoque de lenguaje kernel

1. Definir un lenguaje núcleo (simple y pequeño).
2. Definir una traducción de un lenguaje mas amplio al lenguaje kernel.

- *Abstracción linguística*: abstracción y construcción sintáctica (ej: function).
- *Adornos sintácticos (syntactic sugar)*: notaciones convenientes (shortcuts).

Características del modelo declarativo

Memoria de asignación única

- Variables declarativas: una vez ligada a un valor, esto no cambia y es indistinguible de su valor. Diferencia con las variables de Haskell o ML.
- Valores parciales: estructura de datos (ej:record) que contiene variables no ligadas.

Identificadores de variables

- Son las “variables” de programa, los cuales están asociados a variables de la memoria de asignación única (memoria inmutable).

- *Ambiente*:conjunto de asociaciones de *identificadores* y *variables*. Ej.

Ligadura (Binding)

El operador = realiza una *unificación* entre sus operandos.

- *Simétrica*: ej: $X = f(i : Y \ j : Z) \equiv f(i : Y \ j : Z) = X$
- Si los operandos son iguales, la operación no tiene efecto
- Si los valores parciales son incompatibles, se produce una excepción.

Creación de Valores

La operación básica sobre la memoria de asignación única es la creación de valores.
Ej. $X = 25$.

- Números (enteros y reales). Los *caracteres* serán representados como números (codificación ASCII o Unicode).
- *Registros*: ej: *person*(*name* : "*George*" *age* : 25). Con los registros se pueden representar las listas y tuplas. Ejemplos (pag. 41).
- *Procedimientos*. Otorga flexibilidad.

Un Programa Ejemplo

```
local Factorial in
    Factorial = proc {$ N ?R}
        if N == 0 then R = 1 else R = N * { Factorial N - 1 } end
    end
    local F in
        F = {Factorial 5} – invocación a Factorial(5)
        {Browse F}
    end
end
```

El lenguaje kernel declarativo

Sintaxis (sentencias)

$$
\begin{array}{lll}
< s > & ::= & \textbf{skip} \\
& | & < s >_1 < s >_2 \\
& | & \textbf{local} < x > \textbf{in} < s > \textbf{end} \\
& | & < x >_1 = < x >_2 \\
& | & < x > = < v > \\
& | & \textbf{if} < x > \textbf{then} < s >_1 \textbf{else} < s >_2 \textbf{end} \\
& | & \textbf{case} < x > \textbf{of} < \textit{pattern} > \textbf{then} < s >_1 \\
& & \textbf{else} < s >_2 \textbf{end} \\
& | & \{< x > < y >_1 \ldots < y >_n\} \text{ (procedure call)}
\end{array}
$$

El lenguaje kernel declarativo (cont.)

Sintaxis (valores)

$< v >$	::=	$< number > \mid < record > \mid < procedure >$
$< number >$	::=	$< int > \mid < float >$
$< record >$,		
$< pattern >$	::=	$< literal >$
	$\mid$	$< literal > (< feature >_1 : < x >_1 \ldots$
		$< feature >_n : < x >_n)$
$< procedure >$	::=	**proc** $\{\$ < x >_1 \ldots < x >_n\} < s >$ **end**
$< literal >$	::=	$< atom > \mid < bool >$
$< feature >$	::=	$< literal > \mid < int >$
$< bool >$	::=	**false** | **true**

Notas

- Un programa es una *sentencia*
- El lenguaje usa *tipado dinámico*

Máquina abstracta

- Φ: *Store de única asignación*. Es un conjunto de variables ligadas o no a valores. Ej: $\{x_1, x_2 = x_3, x4 = 10\}$.
  (x_1 no está ligada a un valor, x_2 y x_3 son iguales pero no ligadas a valores, x_4 está ligada al valor 10)
- E: **Ambiente**: mapping de identificadores de variables a entidades de *Psi*.
  (ej: $\{X \rightarrow x_1, Y \rightarrow x_2\}$)
- $(< s >, E)$: es una *sentencia semántica*
- (ST, Φ) es un *estado de ejecución* donde ST es una pila de *sentencias semánticas*
- Una *computación* es una secuencia de estados de ejecución desde el estado inicial.
  $(ST_0, \Phi_0) \rightarrow (ST_1, \Phi_1) \ldots (ST_n, \Phi_n)$

Ejecución de programas

Ejecución de un programa $< s >$

1. El estado de ejecución inicial es $([(< s >, \emptyset)], \emptyset)$, es decir que el stack tiene un único elemento (con ambiente vacío) y el *store* es vacío
2. En cada paso, el tope del stack es tomado y realiza la acción correspondiente a la sentencia
3. La ejecución finaliza (normalmente) cuando se vacía la pila
4. Una pila semántica puede estar en uno de los siguientes estados:
   - *Runnable*: es posible realizar un próximo paso
   - *Terminated*: el stack está vacío
   - *Suspended*: el stack no está vacío, pero no se puede avanzar

Semántica de sentencias

- (**skip**, E) (skip): pop del stack
- $(< s >_1 < s >_2), E)$ (composición secuencial)
  1. push $(< s >_2, E)$
  2. push $(< s >_1, E)$
- (**local** $< x >$ *in* $< s >$ **end**, E): (creación de variable)
  1. crear una nueva variable x en el store Φ
  2. push $(< s >, E + \{< x > \rightarrow x\})$
- $(< X >_1 = < X >_2, E)$: variable-variable binding
  $Bind(E(< X >_1), E(< X >_2))$ en el store

Semántica de sentencias (cont.)

- $(< X >=< v >, E)$: creación de valores
  1. construir el valor representado por $< v >$ y hacer que $E(X)$ refiera a éste
  2. todos los identificadores de v se reemplazan por contenidos en Φ según E (excepto para valores que representen procedimientos)
  3. en el caso que
     $< v >=$ **proc** $\{\$ < y_1 > \ldots < y_n > \$\} < s >$ **end**
     hacer push(**proc** $(\{\$ < y_1 > \ldots < y_n > \$\} < s >$ **end** , $CE)$
     donde $CE = E\,|_{\{<z_1>,\ldots,<z_n>\}}$ donde $< z_1 >, \ldots, < z_n >$ son las *variables libres en* $< s >$
  4. $Bind(E(< X >), < v >)$ en el store

Semántica de sentencias (cont.)

- (**if** $< x >$ **then** $< s >_1$ **else** $< s >_2$ **end**, E): condicional
  1. Si $E(< x >)$ es **true**, push $(< s >_1, E)$
  2. Si $E(< x >)$ es **false**, push $(< s >_2, E)$
  3. Si $E(< x >)$ es indeterminado el stack pasa a modo *Suspended*
- (**case** $< x >$ **of** $< lit >$ $(< f >_1 : < x >_1 \ ...)$ **then** $< s >_1$ **else** $< s >_2$ **end**, E): pattern matching
  1. Si $E(< x >)$ es indeterminado el stack pasa a *Suspended*
  2. Si $label(E(< x >)) =< lit >$ y $arity(E(< x >)) = [f_1 ... f_n]$, push $(< s >_1, E + \{< x >_1 \rightarrow E(x).f_1, ...\})$ sino, push $(< s >_2, E)$
- $(\{< x >< y >_1 \ ... \ < y >_n\}, E)$: invocacion procedural
  1. Si $E(< x >)$ es indeterminado el stack pasa a *Suspended*
  2. Si $E(< x >)$ no es un valor de procedimiento, disparar una excepción de error
  3. Si $E(< x >) = (proc \ \{\$ < z >_1 \ ... < z >_n\} \ < s > \ \textbf{end}, E')$ push $(< s >, E' + \{< z >_1 \rightarrow E(< y >_1), ..., < z >_1 \rightarrow E(< y >_n)\})$

Extensiones sintácticas prácticas al lenguaje kernel

- *Listas de variables en declaraciones*: ej: **local** X, Y ... **in**
- *Valores parciales anidados*: en lugar de **local** A, B **in** A="George" B = 25 X = person(name:A age:B) **end** podemos escribir:
  X = person(name:"George" age:25)
- *Inicialización de variables*: ej: **local** X = 10 ...
- *Expresiones*: notación compacta para denotar una secuencia de operaciones que arrojan valores. Ej:
  $X = Y + Z * 2$ es equivalente a
  { times Z 2 T }
  { plus Y T X }
- *Funciones*: **fun** $\{F\ X_1 \ldots X_n\}$ <s> <expr> **end**
  es equivalente a:
  F = **proc** $\{\$\ X_1 \ldots X_n\ ?R\}$ <s> R = <expr> **end**

Características del lenguaje kernel

Tipado dinámico

El chequeo de los tipos de los valores y variables se realiza en tiempo de ejecución.
No permite definir nuevos tipos.

Alcance (scope) estático

Las invocaciones a procedimientos se evalúan en el ambiente de su definición (y no en el de su invocante).

Destrucción implícita de valores

Manejo automático de la memoria (Garbage collection: recolección de bloques de memoria no utilizados)

Programación declarativa

Características

- Los programas son *composicionales*
- El razonamiento sobre programas es *simple* (ej: inducción estructural), gracias a la *transparencia referencial*, es decir no existen *efectos colaterales*: varias evaluaciones de una expresión arroja siempre el mismo valor

Lenguajes de especificación

Características

- Los programas declarativos son *especificaciones* en un sentido formal, pero no práctico.
- Es posible definir un lenguaje declarativo con mayor poder expresivo: *lenguajes de especificación*.
  Problema: Eficiencia
- Son útiles para *razonar sobre programas*.
- Generalmente requieren el uso de *demostradores de teoremas* (automáticos o semi-automáticos).
- Son difíciles de aplicar en proyectos de gran escala.
- Requieren amplios conocimientos en matemática y lógica de los desarrolladores.
- Ejemplos: Z, VDM, Raise, B, LOTOS, ...

Más sobre el modelo declarativo

Unificación

- Formalmente: $e_1 = e_2$ si y sólo si $\exists$ una sustitución s tal que $e_1[s] = e_2[s]$
  Ejemplo: person(name:X age:Z) = person(name:"George" age:Y) ya que si aplicamos $s = \{(X/"George"), (Z/Y)\}$ a ambas expresiones, éstas se igualan (textualmente)
- Operación que adiciona información al *store* (la sustitución define *bindings*)
- es *simétrica*. Ej: pair(fst:Y snd:Z)=X es igual a X=pair(fst:Y snd:Z)
- dos valores parciales pueden unificar.
  Ej: pair(fst:Y snd:25) = pair(fst:30 snd:X)
  genera los bindings: Y = 30 y X = 25
- si los valores son iguales, unificación no hace nada

Unificación e igualdad (entailment check)

Unificación (cont.)

- dos valores pueden *no unificar*
- puede crear estructuras cíclicas. Ej: List = List(head:X tail:List)
- puede unificar estructuras cíclicas.
  Ej: si tenemos X=f(a:X b:_) e Y=f(a:_ b:Y), X = Y crea una estructuras con dos ciclos

Igualdad (entailment check)

Función booleana que toma dos expresiones y chequea si son iguales o no

Unificación: el algoritmo

- El *store* $= \{x_1, \ldots, x_k\} = UB \cup DV$, $UB \cap DV = \emptyset$.
  - UB_i es el cjto de variables *no ligadas (unbound)* e iguales entre sí. Forman un *equivalence set*. $UB = \bigcup UB_i$
  - DV es el cjto de variables *determinadas o ligadas* a valores
- Ej: $\{x_1 = foo(a : x_2), x_2 = 25, x_3 = x_4 = x_5, x_6\}$ tiene dos *equivalence sets*, $\{x_3, x_4, x_5\}$ y $\{x_6\}$. x_1 y x_2 son determinadas.
- Operaciones primitivas:
  - $bind(ES, v)$ liga todas las variables en el *equivalence set* ES con el valor v
  - $bind(ES_1, ES_2)$ une los conjuntos ES_1 y ES_2 en un solo *equivalence set*
- El algoritmo $unify(x, y)$ se define como sigue:

Unificación: el algoritmo $unify(x, y)$

1. Si $x \in ES_x$ e $y \in ES_y$, $bind(ES_x, ES_y)$
2. Si $x \in ES_x$ e y es determinado, $bind(ES_x, y)$
3. Si $y \in ES_y$ e x es determinado, $bind(ES_y, x)$
4. Si x está ligada a $l(l_1 : x_1 \ldots l_n : x_n)$ e y está ligada a $l'(l'_1 : y_1 \ldots l'_m : y_m)$ con $l \neq l'$ o $\{l_1, \ldots, l_n\} \neq \{l'_1, \ldots, l'_m\}$, generar un error
5. Si x está ligada a $l(l_1 : x_1 \ldots l_n : x_n)$ e y está ligada a $l'(l_1 : y_1 \ldots l_n : y_n)$, hacer $unify(x_i, y_i)$, para todo i, $1 \leq i \leq n$

Manejo de valores con ciclos

- El algoritmo anterior entra en un ciclo infinito
- **Corrección**: garantizar que cada llamado a $unify(x, y)$ se haga a lo sumo una vez por cada par de variables.

Entailment and Disentailment checks ($==$ y $=/=$)

Entailment check (operador $==$)

- $X == Y$ es una función lógica que chequea si $Store \models X = Y$ ($Store$ visto como una conjunción de predicados de igualdad).
- **Algoritmo**: chequeo si las estructuras referenciadas por X e Y son equivalentes.
  Esto se conoce como equivalencia estructural

Chequeos en otros lenguajes (Pascal, C, C++, Java, . . .)

- Si son valores simples (incluyendo punteros o referencias) se compara su representación.
- Generalmente los valores estructurados no son comparables o se comparan por operadores definidos por el usuario.

Computación Iterativa

En general una computación iterativa es un transformador de estados

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots S_final$$

Esquema:

fun { Iterate S_i }
    if { is_done S_i } **then** S_i
    else S_{i+1} **in**
        S_{i+1} = { Transform S_i }
        { Iterate S_{i+1} }
end

Computación recursiva

- Características
- Tamaño de Pila creciente.
- Conviertiendo una Computación recursiva en iterativa.

Técnicas de programación declarativa (cont.)

Programación de **Alto orden** y Técnicas Subyacentes

Uso de procedimientos y funciones como parámetros.

- Abstracción Procedural.
- Genericidad. Ejemplo.
- Instanciación.
  Es posible implementar *generadores* o *factorías*. Ejemplo:

```
fun { MakeSort F }
   fun { $ L }
      { Sort L F }
   end end
```

- Embedding (Embebimiento): Los valores de tipo procedimiento se pueden colocar en estructuras de Datos. Usos:
  - Delayed (lazy) evaluation
  - Modules: record conteniendo operaciones
  - Componentes: procedimientos genéricos que toman módulos como entrada y retorna un nuevo módulo
- Currificación.

Manejo de excepciones en OZ

Definición de un *error*

Definimos un *error* como la diferencia entre el comportamiento real del programa y el comportamiento deseado.

Fuentes de un *error*:

- Argumento de un condicional que no es un valor booleano.
- División por cero
- *Ligadura*:Identificadores que no unifican. Etc.

Extensión del lenguaje núcleo

- **try** y **raise**
- Ejemplo.
- Semántica. Se deja como ejercicio definir la semántica.

Lenguajes funcionales

La programación funcional se basa en la definición de funciones totales (en el sentido matemático).

Características Principales

- **Funciones puras**: no tienen estado propio (memoria) ni efecto colaterales.
- **Recursión**.
- **Funciones de Alto Orden**.
- **Orden de Evaluación**.

Fundamentos Teóricos de la Programación funcional

Fundamentos matemáticos: *cálculo* λ

- *variables*: $x, y, \ldots$
- λ *abstracción*: $\lambda x.t$
  equivalente a: **fun** { $ X } <t> **end**
- *aplicación*: $t_1\ t_2$

Semántica operacional (sistema de reescritura de términos)

Computación: $(\beta - \text{reducción})$ aplicación de funciones (abstracciones) a sus argumentos.
$(\lambda x.t_1)t_2 \rightarrow t_1[x/t_2]$ (reemplazo de x en t_1 por t_2)
El término $(\lambda x.t_1)t_2$ se denomina *redex*.
Un término sin *redexes* está en *forma normal*.

Estrategias de evaluación

- *Full beta-reducción*: Cualquier redex puede reducirse en cualquier momento
- *Orden normal (o evaluación perezosa)*: En cada paso se elige el redex de más a la izquierda
  ej: $\underline{id(id(\lambda z.id\ z))} \rightarrow \underline{(id(\lambda z.id\ z))} \rightarrow \lambda z.\underline{id\ z} \rightarrow \lambda z.z$
- *call-by-name*: No se permiten reducciones dentro de abstracciones.
  ej: $\underline{id(id(\lambda z.id\ z))} \rightarrow \underline{(id(\lambda z.id\ z))} \rightarrow \lambda z.id\ z$
- *call-by-value*: Se elige el redex más exterior y sólo cuando su operando está en forma normal. ej:
  $\underline{id(id(\lambda z.id\ z))} \rightarrow \underline{(id(\lambda z.id\ z))} \rightarrow \lambda z.id\ z$
- *Orden Aplicativo*: en cada paso se elige el redex mas interno y más a la derecha. Esta estrategia es *estricta* o *ansiosa* ya que los argumentos se evalúan siempre (aunque no se utilizen).
- *call-by-need (evaluación lazy)*:, igual que el orden normal, pero los términos duplicados se evalúan solo una vez.

Programación en *cálculo-*λ

Múltiples argumentos

El *cálculo-*λ no provee múltiples argumentos.
Currying: $\lambda(x, y).s \equiv \lambda x.\lambda y.s$

Church booleans

$true = \lambda t.\ \lambda f.\ t$
$false = \lambda t.\ \lambda f.\ f$
ejemplo: $if = \lambda c.\ \lambda s_1.\ \lambda s_2.\ c\ s_1\ s_2$

Numerales

$0 = \lambda s.\ \lambda z.\ z$
$1 = \lambda s.\ \lambda z.\ s\ z$
$2 = \lambda s.\ \lambda z.\ s\ (s\ z)$
...

Programación en *cálculo*-λ (continuación)

Pares

$pair = \lambda f.\ \lambda s.\ \lambda b.\ b\ f\ s$
$first = \lambda p.\ p\ true$
$second = \lambda p.\ p\ false$

Ejemplo: $first(pair\ v\ w) \stackrel{*}{\Longrightarrow} v$
$first(pair\ v\ w) = first(\underline{(\lambda f.\lambda s.\lambda b.\ b\ f\ s)\ v}\ w)$
$\rightarrow first(\underline{(\lambda s.\lambda b.\ b\ v\ s)\ w})$
$\rightarrow first(\lambda b.\ b\ v\ w) = \underline{(\lambda p.p\ true)(\lambda b.\ b\ v\ w)}$
$\rightarrow \underline{(\lambda b.\ b\ v\ w) :\ \ true}$
$\rightarrow true\ v\ w \stackrel{*}{\Longrightarrow} v$

Programación funcional: variedades

- El lenguaje kernel definido, para que sea un lenguaje funcional puro, debe ser restringido con inicialización obligatoria de variables y sólo con declaraciones de funciones es al estilo de *Scheme*.
- Existen lenguajes estáticamente tipados: *Haskell*, *ML*, *Miranda*, . . .
- El modelo declarativo dado es *ansioso (eager)*. *Haskell* utiliza evaluación *perezosa (lazy) o por necesidad (by need)*.
- Algunos garantizan la *transparencia referencial*: permite reemplazar invocaciones a funciones por su definición (previo binding de parámetros), lo cual simplifica el razonamiento sobre programas.
- Razonamiento sobre programas: inducción matemática y estructural
- Alto orden: reciben otras funciones como parámetros

LISt Processing (LISP)

- Es un lenguaje interactivo. Utiliza un intérprete muy simple.
- Adecuado para realizar manipulaciones simbólicas y sobre estructuras de datos.
- Lenguaje aplicativo, ansioso, orientado a expresiones, basado en recursión.
- No es funcional puro: tiene comandos de asignación y condicional.
- Existen varios dialectos: COMMON LISP, SCHEME, ...

Expresiones

- **Símbolos:** son *identificadores de variables*
- **Listas:** están delimitadas por paréntesis. Ej: (f x y).
- **Literales (átomos):** números, caracteres y strings (listas de caracteres). Ej: 0.42, 'c', "una cadena".

LISP (continuación)

Ciclo del intérprete

El intérprete de LISP está en un ciclo en el cual realiza los siguientes pasos:

1. Lectura de una expresión y construcción de su representación.
2. Evaluación.
3. Impresión del resultado de la evaluación.

Ejemplo:

```
> (+ 1 2 3 4 5)
15
>
```

expresiones tildadas (quote expressions)

Una expresión de la forma `quote exp` o `' exp` causa que `exp` no se evalúe.

LISP (continuación)

Expresiones simbólicas (S-Expressions)

1. Un átomo es una *s-expression*.
2. Si *a* y *b* son *s-expressions*, entonces el *dotted pair (a . b)* es una *s-expression*.

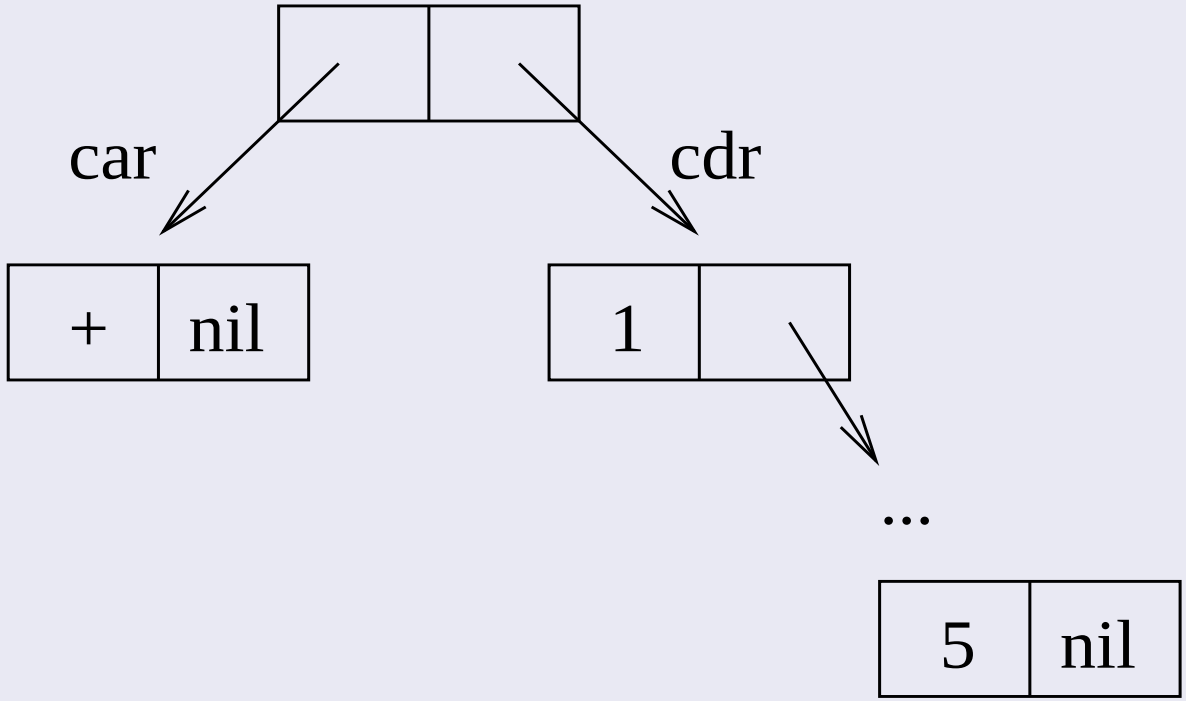
Cada objeto compuesto está representado por un *dotted pair*, construido por *cons*.
Ejemplo: (+ A B) se representa como
(+ . (A . (B . NIL)))

Punteros tipados

Una *s-expression* se representa con un par de punteros con información del tipo de objeto apuntado.
Comunmente se usan los 4 bits altos para denotar el tipo.

LISP (continuación)

Representación de s-expresiones (ej: (+ 1 2 3 4 5))



Funciones sobre listas (CAR y CDR)

(CAR (X0 . . .)) –> X0
(CDR (X0 X1 . . . Xn)) –> (X1 . . . Xn)

LISP (continuación)

Lambda expresiones (funciones anónimas)

Forma: (lambda arg-list expression)
Ejemplo: (defun adder (delta) (lambda (x) (+ x delta)))

Definición de funciones

Forma: (defun function-name arg-list expression)
ejemplo: (defun factorial (n) (if (<= n 1) 1 (* n (factorial (- n 1)))))

Variables (referencias)

Forma: (setf identifier expression)
Ejemplos: (setf f (adder 10)) (funcall f 7)

Haskell

Su nombre es en honor al famoso lógico-matemático Haskell Curry (1900-1982).

Características

- Lenguaje funcional *puro* (sin efectos colaterales).
- Evaluación lazy.
- Estáticamente tipado, con un sistema de tipos avanzado.
- Inferencia de tipos automático (excepto en ciertos casos especiales).
- Soporta polimorfismo paramétrico.
- Definiciones por casos por patrones (pattern matching).
- Definiciones de listas por comprensión.

Haskell (continuación)

Valores y tipos de datos

- La evaluación (reducción) de expresiones produce *valores*.
- Todos los valores son de un tipo determinado.
- Valores compuestos (estructurados):
  - Listas (homogéneas: valores del mismo tipo)
  - Tuplas (heterogéneas)

Polimorfismo paramátrico

- Variables de tipo.
- Las variables de tipo están cuantificadas (implícitamente) universalmente.

Ejemplo: La función
length :: [a] -> Integer
toma familias de listas de tipos *a* en enteros.

Haskell (continuación)

Tipos definidos por el usuario

Sintaxis: data name [type-parameters]= data-constructor
Ejemplos:
data Bool = False | True
data Point a = pt a a
data Tree a = Leaf a | Node (Tree a) (Tree a)

Sinónimos de tipos

Sintaxis: type type_name = type_def
Ejemplos:
type String = [Char]
type Person = (Name,Address)

Haskell (continuación)

Listas por comprensión

Sintaxis: [data | generators , optional_guards]

Ejemplos:

```
quicksort []     =  []
quicksort x:xs   =  quicksort [ y | y < xs, y < x ]
                    ++ [x] ++
                    quicksort [ y | y < xs, y >= x ]
```

Listas infinitas

Gracias a la evaluación lazy es posible definir estructuras infinitas

Ejemplo:

```
even = [ n | n mod 2 = 0 ]
```

ML

Su nombre es en honor al famoso lógico-matemático Haskell Curry (1900-1982).

Características

- Lenguaje funcional con características imperativas (referencias).
- Evaluación estricta (algunas implementaciones permiten lazy).
- Estáticamente tipado, con un sistema de tipos avanzado.
- Inferencia de tipos automático (excepto en ciertos casos especiales).
- Soporta polimorfismo paramétrico.
- Definiciones por casos por patrones (pattern matching).
- Definiciones de listas por comprensión.

Capítulo 5 - Paradigma Relacional

Programación Relacional

- Que es la Programación Relacional.
- El Modelo de Computación Relacional.
- Implementación del modelo de computación relacional.
- Relación con la programación logica.
- Lenguaje de programación relacional PROLOG.

Paradigma Relacional... cont.

Relación vs. Funciones

- Cero, una o mas salidas.
- Para el mismo set de valores de entrada -> Salidas diferentes.

Aplicaciones

- Bases de Datos Deductivas.
- Sistemas Expertos.
- Procesamiento de Lenguaje humano.

- Ventajas y Desventajas.

Paradigma Relacional... cont.

Extensión de la programación declarativa

- *choice* $< s >_1$ [] . . . [] $< s >_n$ *end*. Elige nodeterministicamente una de un set de alternativas
- *fail*. Indica que la alternativa es erronea.

Un Programa Ejemplo

```
fun {Soft} choice beige [] coral end end
fun {Hard} choice mauve [] ochre end end
proc {Contrast C1 C2}
    choice C1 = {Soft} C2 = {Hard} [] C1 = {Hard} C2 = {Soft} end
end
fun {Suit}
     Shirt Pants Socks
in
    {Contrast Shirt Pants}
    {Contrast Pants Socks}
    if Shirt==Socks then fail end
    suit (Shirt Pants Socks)
```

Arbol de Búsqueda

La estrategia de búsqueda puede ser representada gráficamente con un árbol de búsqueda.

Control sobre la búsqueda (Búsqueda Encapsulada)

Controlar cuales elecciones deben ser hechas y cuando.

- Estrategia depth-first-search, breadth-first-search, u otra.
- Se debería poder especificar cuantas soluciones son calculadas: solo una, todas, o bajo demanda.
- Es importante para la *modularidad* y *composicionalidad* (busqueda encap. se ejecuta dentro de otra).

Se logra con la búsqueda encapsulada. El programa relacional se ejecuta dentro de un ambiente (tenga en cuenta que hay multiples *binding* de la misma variable, cuando diferentes elecciones son hechas).

Extendiendo el Modelo: Función **Solve**

Provee búsqueda encapsulada.
{Solve F}: retorna una solución al programa relacional **F**, como una lista *lazy* de todas las soluciones.
Ej. L = {Solve **fun** {$} **choice** 1 [] 2 [] 3 **end end** }, retorna la lista *lazy* [1 2 3].
La función *SolveOne* y *SolveAll* se define en base a *Solve*.

Programación Lógica

- Impacto en la década del 70.
- La programación Lógica tiene 2 semánticas: una lógica y una operacional.
- Elementos de un lenguaje de programación lógico (*axiomas, query, probador de teoremas*). Características de los Probadores de teoremas (limitados, eficiencia, pruebas constructivas).

Relación del Modelo Declarativo con la Programación Lógica

Traducción de un programa declarativo a una sentencia de la lógica (ver cuadro de la figura 5.2 del apunte).

Ejemplo (Append Determinístico)

```
proc {Append A B ?C}
   case A of nil then C=B
      [] X|As then Cs in
         C=X|Cs
         Append As B Cs
   end
end
```

$\forall a, b, c : append(a, b, c) \Leftrightarrow (a = nil \wedge c = b) \vee (\exists\, x, a', c' : a = x|a' \wedge c = x|c' \wedge append(a', b, c'))$

Que pasa si hago las siguientes consultas?:*Append [1 2 3] [4 5] X*, *Append X [3] [1 2 3]* y *Append X Y [1 2 3]*

Características de Prolog

- Ideados a principio de los años 70.
- Es un lenguaje Interpretado.
- Prolog usa *cláusulas de Horn* como sintaxis con una semántica operacional basada en el principio de resolución.
- el probador de teoremas usa las *cláusulas de Horn* y las ejecuta basandose en una regla de inferencia: *modus ponens*
- Programación Higher-order no es soportada en pure Prolog.
- Su ejecución se basa en dos conceptos: la *unificación* y el *backtracking*.

Cláusulas de Horn

Constituyen reglas del tipo “Modus Ponens“. El antecedente se denomina secuencia de objetivos.

$\forall x_1, x_2, ..., x_k . A_1 \wedge A_2, ..., A_n \rightarrow A$

Significa que para probar A primero se debe probar A_1,..., A_n.

Términos de Prolog

- *Símbolos de Variables*. Comienzan siempre cn mayúsculas (*L, L2, Persona*). Estas se ligan cuando existe algún objeto representado por ella.
- *Predicados*. Ej.: $padre\_de(juan, berta)$, $ama(maria, X)$.
- *Atomos*: constantes y Variables. Los predicados también son átomos (cerrados cuando todas sus variales están ligadas).
- *Listas*.Ej.: [] (lista vacía (*nil*), $[A, B, C]$, $[Cabeza|Resto]$.

Cláusulas de Prolog

Las cláusulas son estructuras que permiten representar *hechos*, *preguntas o consultas* y *reglas*.
Sintaxis:

- **Reglas**: $cabeza :- cuerpo$. Logicamente es:
  $cuerpo \rightarrow cabeza$.
  Ej 1. "A es hijo de B, si B es padre de A y B es hombre.“
  hijode(A,B) :- padrede(B,A), es_hombre(B).
  Esta regla representa la fórmula:
  $\forall AB.((es\_padre(B,A) \wedge es\_hombre(B)) \rightarrow es\_hijo(A,B))$
  Ej 2 "A es abuelo de B, si A es padre de C y C es padre B.”
  abuelode(A,B) :- padrede(A,C), padrede(C,B).
- **Hechos**: $padre(juan, maria)$, $padre(juan, maria) :- true$.
  Hechos con variables: $suma(0, X, X)$.
- **Consultas**: Ej. ”pablo es abuelo de maria¿‘
  ?- abuelode(’pablo’, ’maria’). ?- abuelode(X, ’maria’).

Unificación

Definición Previa: Una **sustitución** es un conjunto finito de pares de la forma $\{v1 \rightarrow t1, v2 \rightarrow t2, \ldots, vn \rightarrow tn\}$, donde cada vi es una variable, cada ti es un término (distinto de vi) y las variables vi son todas distintas entre sí.

Composición de Sustituciones

Dadas dos sustituciones s y s', se define la operación de composición $s \bullet s'$ de la siguiente forma:
Si $s = \{v1 \rightarrow t1, v2 \rightarrow t2, \ldots, vn \rightarrow tn\}$ y
$s' = \{w1 \rightarrow t'1, w2 \rightarrow t'2, \ldots, wn \rightarrow t'n\}$, entonces
$s \bullet s' = \{v1 \rightarrow t1s', v2 \rightarrow t2s', \ldots, vn \rightarrow tns', w1 \rightarrow t'1, w2 \rightarrow t'2, \ldots, wn \rightarrow t'n\} - \{vi \rightarrow tis' | vi = tis'\} - \{wj \rightarrow t'j | wj \in \{v1, \ldots, vn\}\}$

Definición de Unificación

Un *unificador* de dos átomos $c1$ y $c2$, es una sustitución s tal que $c1\ s = c2\ s$.
Ej. Dado $c1 = p(a, Y)$ y $c2 = p(X, f(Z))$. Los siguientes son unificadores de $c1$ y $c2$:

- $s1 = \{X \to a, Y \to f(a), Z \to a\}$ ya que $c1\ s1 = c2\ s1 = p(a, f(a))$.
- $s2 = \{X \to a, Y \to f(f(a)), Z \to f(a)\}$ ya que $c1\ s2 = c2\ s2 = p(a, f(f(a)))$.

Un unificador más general (*umg*) de dos átomos $c1$ y $c2$ es el unificador g de ambos tal que, cualquier unificador s de $c1$ y $c2$ se puede obtener como la composición de g con alguna otra sustitución u, es decir $s = g \bullet u$.
En el ejemplo anterior el *umg* de $c1$ y $c2$ sería: $g = \{X \to a, Y \to f(Z)\}$ con $c1\ g = c2\ g = p(a, f(Z))$.

Algoritmo para encontrar el *umg* (ver apunte), basado en la noción de 'conjunto de disparidad'.

Fundamentos Lógicos

Los predicados y funciones se denominan *literales* (pueden ser negados o no).
Conectivas lógicas necesarias: *negación* ($\neg$), *disyunción* ($|$), y *conjunción*.
$A \rightarrow B$ puede definirse como $\neg A|B$.
$ABUELO(X, Y) : -PADRE(X, Z), PADRE(Z, Y)$ es lógicamente equivalente a (por la regla anterior y por ley de morgan) $\neg PADRE(z, y)|\neg PADRE(x, z)|ABUELO(x, y)$

Principio de Resolución de Robinson

El mecanismo de inferencia se basa en el principio de resolución de Robinson.
Utiliza un sola regla de inferencia : regla de resolución.

$$\frac{\{B_1,...,B_k,A_1,...,A_m\}\{D_1,...,D_l,C_1,...,C_n\}}{\{A_1,...,A_m,C_1,...,C_n\}\sigma}$$

donde σ es el *umg* de $\{B_1, ..., B_k\}$ y $\{D_1, ..., D_l\}$.

Prolog: Principio de Resolución

Regla de resolución

Hay dos casos particulares de esta ley:

- $\neg A \vee B$ y de A se deriva B (modus ponens).
- de $\neg A$ y de A se deriva la clausula vacía ([]).

Prinicipio de Resolución: características

- Utiliza el método de reducción al absurdo.
- Comienza negando lo que queremos probar.
- Obtiene nuevos resolventes aplicando la regla de resolución.
- Si se consigue llegar a la cláusula vacía, se habrá probado el objetivo.

Ejemplo

papa(juan, marta).
recien_nacido(marta).
orgulloso(X):-padre (X,Y), recien_nacido(Y).
padre(X,Y):- papa(X,Y).
padre(X,Y):- mama(X,Y).
.
Consulta: *?orgulloso(X).*
.
Previamente, convertir las *cláusulas de Horn* a expresiones lógicas disyuntivas (formal normal disyuntiva).

Estrategia de Resolución usada en Prolog

Mecanismo

- Partiendo de la consulta (objetivo), ejecutar paso a paso la regla de resolución para obtener nuevos objetivos, hasta llegar a la pregunta vacía (corresponde a un *éxito*).
  O una pregunta para la cual no exista resolvente con ninguna sentencia (corresponde a un *fracaso*).
- Esta estrategia se denomina **estrategia de resolución SLD**.
- La secuencia de pasos desde la pregunta original hasta la pregunta vacía se denomina refutación SLD.

Espacio de Búsqueda

- Recorrido *depth-first-search*.
- Arbol SLD.
- Ejemplo. *?orgulloso(X).*

El predicado *cut (!)* y el problema de la Negación

El predicado *cut* (!)

- Es una notación que nos permite podar el Arbol SLD. Se lo utiliza para cortar el *backtracking*.
- Es de caracter extra-lógico (no es un predicado estandar de la lógica).
- Se encuentra presente por cuestiones de *eficiencia*.
- Debe usarse con cuidado dado que puede podarse una rama de éxito.
- Ejemplo:
  rel :- a, b, !, c.
  rel:- d.

Solo busca la primer solución para *a* y *b*. Para *c* busca multiples soluciones.
En que caso será evaluada la segunda definición de *rel*?.

El problema de la negación

- Los programas expresan conocimiento positivo y no dicen nada acerca del resto.
- Todo aquello que no está en el programa y que no pueda ser derivado a través de sus reglas es FALSO y por lo tanto su negación (metapredicado *not*) es cierta.
- Su uso incorrecto puede resultar peligroso.
- Ejemplo: piloto_suicida(X):- conduce(X), irresponsable(X).
  irresponsable(X) :- not(responsable(X)).

Predicado *fail*

- Se utiliza para forzar el backtracking.
- Se puede definir la negación con el *cut* y el *fail*.
  no(P) :- P, !, fail.
  no(P).

Ejemplos Varios

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
---
factorial(1,1).
factorial(X,Y) :- X > 1, A is X - 1, factorial(A,B), Y is X * B.
---
n_veces(_,0,[]).
n_veces(X,N,X|RS):- n_veces(X,N2,RS), N is N2+1.
n_veces(X,N,Y|RS):- X \== Y, n_veces(X,N,RS).
---
subset([], []).
subset([X|Xs], [X|Ys]) :- subset(Xs, Ys).
subset([_|Xs], Ys) :- subset(Xs, Ys).
Que predicado define?:
predicado([],L,L).
predicado([X|L1],L2,[X|L3]) :- predicado(L1,L2,L3).
```

Programación con estado explícito.

Definición de *estado*

Un estado es una secuencia de valores en el tiempo que contienen los resultados intermedios de una computación específica.

Hasta ahora se venía haciendo un majeno del concepto de estado de manera implícita.

```
fun SumList Xs S
    case Xs of nil then S
        [] X|Xr then SumList Xr X+S
    end
end
```

Si invocamos *{SumList [1 2 3 4] 0}*, obtenemos la secuencia siguiente:
[1 2 3 4] # 0
[2 3 4] # 1
[3 4] # 3
[4] # 6
nil # 10
Esta secuencia es un estado.

Estado explícito

Un estado explícito en un procedimiento, es un estado cuyo tiempo de vida se extiende más allá de una invocación al procedimiento, sin estar presente en los argumentos del procedimiento.

Estado explícito: Características

- Un componente, además de depender de sus argumentos, depende también de un parámetro interno, el cual se llama su ”estado".
- Este parámetro le provee al componente una memoria de "largo plazo".
- Sin estado, un componente solo tiene memoria de corto plazo, memoria que sólo existe durante una invocación particular del componente.

Extensión del modelo declarativo

Extendemos el modelo declarativo con *celdas*.

Características de una Celda

- tiene un nombre,
- un tiempo de vida indefinido,
- y un contenido que puede ser modificado.

Podemos usar una celda para añadir una memoria de largo plazo a *SumList*:

```
local C={NewCell 0} in
    fun {SumList Xs S}
        C:=@C+1
        case Xs of nil then S
            [] X|Xr then {SumList Xr X+S}
        end
    end

    fun {ContadorSum} @C end
end
```

Extensión del modelo declarativo

Operaciones que extienden el modelo declarativo

- {NewCell X C}. Crea una celda nueva *C* con contenido inicial *X*.
- {Exchange C X Y}. Liga atómicamene *X* con el contenido antiguo de la celda *C* y hace que *Y* sea el contenido nuevo.
- @C. Retorna el contenido actual de la celda *C*.
- C:=X. Coloca a *X* como el contenido nuevo de la celda *C*. Además arroja el valor anterior de *C*.

Las ultimas dos se pueden definir en términos de *Exchange* (ejercicio).

Extensión de la Máquina Abstracta

Se extiende con una nueva *memoria mutable*, la cual contiene las celdas, es decir pares *(x : y)*, donde *x* está ligado a un nombre de celda (valor), e *y* puede ser cualquier valor parcial.

Semántica de las operaciones sobre celdas

El Estado de ejecución de la máquina abstracta es ahora una tripla (ST, σ, μ).

- ({NewCell <X> <Y>}, E).
  - Crear un nombre de celda único *n* en la memoria inmutable σ para la celda.
  - Ligar E(<y>) y *n*.
  - Si la ligadura tiene éxito, entonces agregar el par $(E(< y >) : E(< x >))$ a la memoria mutable μ.
  - Si la ligadura falla, entonces lanzar una condición de error.
- ({Exchange <x> <y> <z>},E).
  Si (E<x>) está determinada, entonces:
  - Si E(<x>) no está ligada al nombre de una celda, lanzar una condición de error.
  - Si μ contiene $(E(< x >) : w)$, entonces realizar las acciones siguientes:
    - Actualizar μ con $(E(< x >) : E(< z >))$.
    - Ligar $E(< y >)$ y *w* en σ
  - Si la condición de activación es falsa, entonces suspender la ejecución.

Ejemplos

Ejemplo 1:

```
local X Y Z in
    X=1
    {NewCell X Y}
    {NewCell 1 Z}
    {Browse @Y==@Z}
end
```

Ejemplo 2:

```
local X C in
    {NewCell 2 X}
    {NewCell 1 C}
    X=C
    {Browse @X}
end
```

Que sucede con la ejecución de este último ejemplo?

Aliasing

Dos variables son *alias (sinónimos)* si se refieren a la misma entidad (celda).

Aliasing

- Complica el razonamiento sobre los programas.
- Causa de la aparición de efectos colaterales.
- Principal motivo del porque las abstracciones de datos son adecuadas en el modelo con estado explícito.
- Encapsulamiento del estado. Idea de los TADs (minimiza los efectos colaterales).
- En el modelo declarativo la ligadura de variables produce aliasing: `X=Y`. Otros casos?
- Ejemplo en C:

```
int x = 5;
int *y = &x;
int *z;
z=y;
...
```

Programación basada en Componentes

- **Encapsulamiento**: debería ser posible ocultar los detalles internos de cada parte (information hidding).
- **Composicionalidad**: se deberían poder combinar las partes para formar otras.
- **Instanciación/invocación**: debería ser posible crear varias instancias a partir de una misma definición. Este mecanismo permite la reutilización de partes.

Los componentes de pueden definir a diferentes niveles.

- **Abstracción procedural**: el componente se denomina la definición de procedimientos y sus instancias son las invocaciones o llamados a procedimientos.
- **Functores (unidades de compilación)**: pueden ser compilados en forma separada. Sus definiciones son functores y sus instancias módulos.
- **Componentes concurrentes**: un sistema con entidades independientes, interactuando entre sí por medio de mensajes.

Programación Imperativa

La programación orientada a objetos provee mecanismos para la definición y uso de componentes, con un mecanismo adicional para la definición de nuevos componentes: la *herencia*.

Abstracción Procedural

- Un procedimiento es una abstracción de un comando, es decir una acción parametrizada.
- Una función es una abstracción de una expresión (denota un valor).
- Parametros *formales* y *actuales*.

Abstracción Procedural- Mecanismos de Pasajes de Parámetros

Por copia

- **por valor** (o entrada): el parámetro formal se liga a una copia del parámetro actual. Cualquier modificación al parámetro formal no afecta al parámetro actual.
- **por resultado** (o salida): el parámetro formal se copia al parámetro actual durante el retorno. En la entrada al procedimiento el parámetro formal no está ligado a ningún valor concreto.
- **por valor-resultado** (o entrada-salida): combinación de los dos mecanismos anteriores.

Denotacionales

- **por referencia**: el parámetro formal se convierte en una nueva identidad del parámetro actual. La modificación del parámetro formal afecta al actual.
- **por nombre**: la expresión (sin evaluar) que denota el parámetro actual se liga al parámetro formal. Cada referencia al parámetro formal, dentro de la subrutina, lanza la evaluación de la expresión asociada.
- **por necesidad** (by need) es una modificación del anterior, donde la expresión (función) se evalúa una sola vez.

Lenguajes Imperativos

Expresiones y Comandos

Algunas sentencias en un lenguaje imperativo toman la forma de expresiones o comandos de control de flujo de ejecución.

- Una expresión representa un valor (de algún tipo de datos).
- Los comandos representan construcciones sintácticas de control estructurado del flujo de ejecución de las sentencias que contienen.
- La operación fundamental de cómputo es la asignación y la evaluación de expresiones.

Características de los Lenguajes Imperativos

- Los lenguajes imperativos como Pascal o C, son lenguajes monomórficos.
- Algunos operadores son polimórficos (ej: operador + de Pascal, el cual suma enteros,reales, concatena strings y además hace unión de conjuntos).
- Operadores por Sobrecarga: implementación de varias funciones diferenetes con argumentos de diferentes tipos.

Clasificación de los comandos o sentencias de control

- **Bloque o Secuencia**.Ej. **begin** *s_0*; s_1; . . . *s_n* **end**.
- **Selección**: Ej. **`if`** `<cond>` **`then`** `<s1>` **`else`** `<s2>` **`end`**.
- **Iteración**: Ej. clásicos comandos **while**, o *repeat-until*.
- **Secuenciadores**: Ej. `goto label`.
- **Excepciones**: Una forma muy común de las excepciones es: **try** **catch**(e1:T1) . . . **catch**(en:Tn)

Sentencias relacionadas directamente con el modelo con estado explícito.

Asignación

- C es un lenguaje con un conjunto de sentencias de asignación muy rico.
- Es la única sentencia que realiza un cambio de estado durante la ejecución de un programa.
- Las sentencias de asignación son expresiones (operadores), denotan un valor, lo que permite al programador realizar composición de asignaciones (ver tabla).

Operador	Descripción
=	asignación (ej: x = y + 1)
+=	acumulación (ej: x += y es equivalente a x = x + y)
-=	(ej: x -= y es equivalente a x = x - y)
*=	(ej: x *= y es equivalente a x = x * y)
/=	(ej: x *= y es equivalente a x = x / y)
%=	(ej: x %= y es equivalente a x = x % y)
x++	pos-incremento (ej: x = y++; asigna y a x y luego incrementa y)
x–	pos-decremento
++x	pre-incremento (ej: x = ++y; incrementa y, luego asigna y a x)
–x	pre-decremento

Punteros

- Son semejantes a las *celdas* del modelo declarativo.
- Los punteros son datos básicos del lenguaje. Sus valores representan direcciones de memoria de otras entidades (variables o valores).
- El tipo de datos puntero es un tipo parametrizado ya que son tipados, esto es, un puntero apunta a entidades de un tipo determinado.
- Sintaxis: `T * ptr`, donde `T` es el tipo de los valores apuntados por el puntero `ptr`.
- Tienen 2 operadores asociados:
  - *referenciación (operador &)*: el operador prefijo & aplicado a una variable, retorna su dirección de memoria, la cual puede ser asignado a pun puntero de un tipo compatible con dicha variable.
  - *desreferenciación (operador *)*: el operador (prefijo) `*` aplicado a un puntero `p` retorna el valor apuntado por `p`.

Ejemplo

```
...
{
    int v = 5; int * p1 = &v;
    int p2;
...
    p2 = p1; /* asignacion (copia) entre punteros */
    (*p1)++; /* el valor de v es incrementado (6) */
    printf("El valor de v es: %d
n", *p2);
}
...
```

Vectores y Punteros

- C permite definir vectores semi-dinámicos.
- declaración de la forma `T v[N]`.
- la declaración de un parámetro formal `float v[]` es equivalente a `float *v`.
- la expresión `v[i]` es equivalente a `*(v+i)`.
- es posible realizar operaciones aritméticas sobre punteros (no es equivalente a la aritmética sobre enteros).
  El puntero `v` apunta a su dirección base.
  La expresión `v+i` es equivalente a `address(v)+i*sizeof(int)`.

Capítulo 8 - Manejo de Memoria

Modelo Von Newmann: los datos y setencias de programas están en la misma memoria RAM.

Bloques de memoria de un *proceso* (programa en ejecución)

- **Segmento de código** (text segment): contiene las instrucciones del programa.
- **Segmento de datos**: área de almacenamiento de las variables globales (y variables locales estáticas en C o C++).
- **Stack**: área de almancenamiento en las que se lleva el control de las activaciones de las subrutinas (direcciones de retorno) y en la mayoría de los lenguajes de programación modernos se lleva información de control adicional para acceso a ambientes locales y no locales, además se almacenan las variables locales y valores de los parámetros actuales de las subrutinas activas.
- **Heap**: mantiene bloques de memoria que representan datos del programa creados dinámicamente (*new* (de Pascal, C++, Java, etc), *malloc* de C), o valores creados implícitamente como el lenguaje utilizado aquí o lenguajes funcionales como Haskell o ML.

Manejo de Memoria

Manejo del Stack

Lenguajes antiguos (*Fortran, Cobol, etc.*):

- hacen un manejo estático de la memoria. Asignan en tiempo de compilación un área definida para cada subrutina.

Lenguajes Modernos (Pascal, C, Java, etc):

- hacen un manejo dinámico del Stack.
- solo hacen manejo estático para las variables globales.
- hacen uso del stack para: el control de la invocación y retorno de subrutinas, el almacenamiento de var. locales, parametros , etc.

Manejo de Memoria

Cada vez que una subrutina es invocada se crea un *registro de activación* (ver figura 8.3 del apunte).
Cuando finaliza la ejecución de una subrutina se elimina del stack el registro de activación correspondiente.

Elementos para el manejo del stack

Formato de un Registro de activación:

- valores temporales.
- variables locales.
- *dynamic link*: apunta a una dirección base del registro inmediato anterior. Esto implementa un Stack como una lista enlazada.
- *static link*: apunta a la dirección base del registro que lo contiene estaticamente (solo es utilizado en lenguajes con alcance estático).
- dirección de retorno. Apunta a la próxima setencia que debe ejecutarse cuando finalize la subrutina corriente.
- parámetros.
- valor de retorno (solo en caso de funciones).

Manejo de Memoria

Elementos para el manejo del stack

Registros del CPU:

- *Stack Pointer (SP)*. Apunta a la dirección base del registro de activación corriente.
- *Frame Pointer (FP)*. Apunta al campo *dynamic link* del registro corriente.

Ejemplo.

Implementación del manejo de alcance de ambientes

- En un lenguaje con alcance dinámico (*dynamic binding*), el acceso a una variable o es local al bloque corriente, o puede accederse siguiendo el *dynamic link*
- En un lenguaje con alcance estático hay 2 posibles implementaciones:
  - siguiendo el *static link* (representa el conjunto de ambientes como una lista enlazada),
  - a través de un *display* (pag. 156 del apunte).

Manejo de Memoria

Valores creados dinamicamente. Manejo del HEAP

Datos del programa creados dinámicamente (*new* de Pascal, C++, Java, etc, *malloc* de C),

- Valores referenciados por *referencias* (JAVA, EIFFEL), o *punteros* (C, Pascal, C++).
- En lenguajes como JAVA, OZ, EIFFEL, etc., la destrucción es responsabilidad del recolector de basura o de un mecanismo *contador de referencias*.
- En lenguajes como Pascal, C y C++ el programador es responsable de la creación y destrucción de estos valores.

  Consecuencias:

  - **Basura**: un valor en el HEAP no tiene referencias. Ejemplo.
  - **Referencias colgadas**: se intenta acceder a un valor que ha sido destruido. Ejemplo.

Manejo de Memoria

Manejo del HEAP

Está implementado en las operaciones de *allocation* y *deallocation* de bloques en el heap. Contiene 2 listas:

- lista de bloques asignados.
- lista de bloques libres.
  Bloques:
- tamaño Fijo (LISP, Haskell).
- tamaño variables (Pascal, C, Java, Eeiffel,...). Acarrea el problema de la *fragmentación del Heap*.
  Soluciónes al problema de la fragmentación:
  - **Compactación total**: reconvierte los múltiples bloques libres en uno solo (*costo*: implica modificar las referencias ya que bloques fueron movidos).
  - **Compactación parcial**: esto se realiza en la operación de liberación de un bloque. Al momento de liberar un bloque, si éste tiene bloques adyacentes libres, se fusionan en un solo bloque. Esto es mas eficiente y generalmente en la práctica funciona muy bien.

Manejo de Memoria

Manejo Automático del HEAP

Evita los errores mencionados anteriormente.
Hay básicamente dos métodos implementados en los lenguajes de programación modernos.

- **Contadores de referencias**: cada bloque de memoria que representa un valor del programa tiene asociado un contador de referencias al bloque. Simple de implementar, pero genera una sobrecarga en la ejecución de un programa.
- **Recolectores de basura**: en este enfoque el mecanismo permite que el programa vaya generando basura y en algún momento de la ejecución se dispare el *recolector de basura* al cual se encargará de detectar los bloques basura para luego recuperarlos.

Algoritmos de recolección de Basura

- **Mark and sweep**.
- **Generacionales**.
- **Copia**.

Capítulo 9 - Programación Orientada a Objetos

Estudio de:

- **Objetos**.
- **Clases**.
- **Herencia**.
- **Dynamic Binding**.
- **Polimorfismo**.
- **Genericidad**.

Objetos

Una función (o varias) con una memoria interna (estado) se denomina un *objeto*. Una primer versión de la implementación de un *objeto*:

```
local C in
        C={NewCell 0}
        fun {Inc}
                C := @C + 1
                @C
        end
        fun {Read}
                @C
        end
end
```

Programación Orientada a Objetos

Encapsulamiento

- **Interface** (funciones **Inc** y **Read**).
- **Implementación** (oculta).

CLASES

- Las clases juegan un rol de fábricas de objetos.
- Definen un **módulo**.

Programación Orientada a Objetos

Clases

Posible implementación:

```
declare fun {NewCounter}
    local C Inc Read in
        C={NewCell 0}
        fun {Inc}
            C := @C + 1
            @C
        end
        fun {Read}
            @C
        end
        counter(inc:Inc read:Read)
    end
end
```

Utilización

C1 = {NewCounter}; C2 = {NewCounter}; {Browse {C1.inc}}; {Browse {C2.read}}

Extensión del Lenguaje Núcleo

Ver Figura 9.3 (apunte de la materia).
Ejemplo:

```
class Counter
    attr val;
    meth init(Value)
        val := Value
    end
    meth inc(Value)
        val := val + Value
    end
    meth get
        @val
    end
end
```

Ejemplo de Uso

C = {**New** Counter init(0)}; {C inc(6)}; {Browse {C get}}

Implementación de la clase **Counter** en el lenguaje núcleo con estado

```
local
    proc {Init M S}
        init(Value)=M in (S.val) := Value
    end
    proc {Inc M S}
        inc(Value)=M in (S.val) := @(S.val) + Value
    end
    proc {Get M S}
        get = M in @(S.val)
    end
in
    Counter = c(attrs:[val] methods:m(init:Init inc:Inc get:Get))
end
```

Similitud en otros lenguajes

- Función **New** para crear instancias.
- **{C inc(x)}** es una aplicación sobre el método inc(x) del objeto C. (Java, Eiffel, C++ : `object.method`)

Definición del Operador **New**

```
fun {New Class Init}
   Fs = {Map Class.attrs fun {$ X} X#{NewCell _} end}
   S = {List.toRecord state Fs}
   proc {Obj M}
      {Class.methods.{Label M} M S}
   end
in
   {Obj Init}
   Obj
end
```

Clases

- Los objetos son procedimientos.
- Es un procedimiento que relaciona un mensaje con un procedimiento que implementa el método correspondiente.
- El estado del objeto está oculto.

Partes de las Clases

- **Atributos**. es una celda que almacena parte del estado del objeto. Cada instancia tiene sus propios atributos. Excepciones: *Atributos de clases*: en Java o C++ definidos por el modificador **static**. Algunos lenguajes POO tienen otros modificadores en las declaraciones de las clases, atributos y métodos.
- **Métodos**. Declarados por la palabra **meth**.
- **Propiedades**. Declarados por la palabra **prop**. Ej. **final**.

Inicialización de atributos

- **por instancia**.
- **por clase**.

Programación Orientada a Objetos

Métodos y Mensajes

Una aplicación {Obj M} puede hacerse de dos formas:

- **M es un record estático**. Ej. {Counter inc(6)}
- **M es un record dinámico**. Ej. {Obj M}, M es una variable.

Otras características

- **Métodos con un número fijo o variable de argumentos**.
  ```
  meth foo(a:A b:B ...) = M
      % body
  end
  ```
- **Argumentos opcionales**.
  ```
  meth foo(a:A b:B<=5)
      % body
  end
  ```
- Un método puede ser **privado** dentro de la clase (oculto fuera de la clase).
- El nombre de un método puede computarse dinamicamente.
  ```
  meth !A(x) % body end
  ```

Programación Orientada a Objetos

Herencia

- Define nuevas clases a partir de otras.
- Define una relación entre clases (transitiva y no reflexiva).

Control de Acceso a métodos. Ligadura Estática o Dinámica.

```
class A {
    ...
    meth m
        {self m1 ...}
    end

    meth m1
        ...
    end
    ...
end
```

```
class B from A
        ...
        meth m1
            ...
            {A.m ...}
        end
        ...
end
```

Suponer las sentencias `O = {New B ...} ; {O m}`. A que versión de `m1` debe invocar `m`.

Control de Acceso a métodos. Ligadura Estática o Dinámica.

Operadores de Alcance:

- Operador **self**. JAVA y C++: **this**. Eiffel: **current**.
- Operador **,**. Ej {A,m1 ...}. Java:**super.m1**, C++: **clase::m1**.

Implementación del alcance dinámico

Métodos Virtuales.

- Invocaciones se resuelven en tiempo de ejecución.
- Cada clase tiene asociada un vector de punteros a los métodos que requieran *dynamic binding*.
- Cada objeto contiene (además de su estado) un puntero a dicha tabla.
- Cada invocación dinámica se resuelve con un acceso extra a la tabla de punteros.

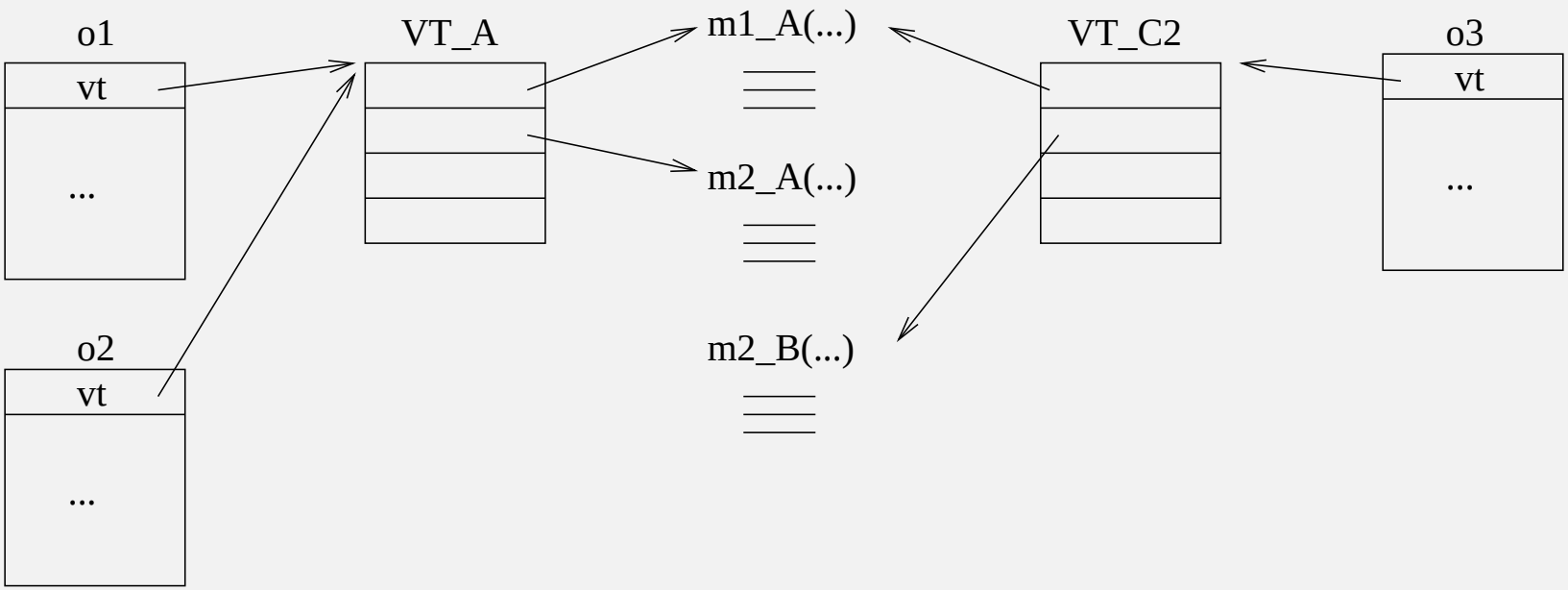


Figure: Tablas Virtuales.

Características del método

- Utilizado en JAVA, Eiffel, C#, C++, etc.
- Problemas: tamaño de las tablas en grandes sistemas. Herencia Múltiple.
- Operador **virtual** de C++.
- Operador **final** de JAVA

Programación Orientada a Objetos

Control de acceso a elementos de una clase

Control de la visibilidad.

- `public` es visible para el resto del programa
- `private` es visible sólo dentro de la clase (o instancia).
- `protected` es visible dentro de la clase y de sus clases derivadas.

Polimorfismo Basado en Herencia (por inclusión)

Define la herencia en términos de sistema de tipos, en base a una relación tipo-subtipo.

- Requiere que sus operaciones tomen referencias o punteros a objetos.
- Un operador toma una referencia a un tipo o cualquiera de sus derivados.
- Es necesario: *binding dinámico*.

Clases y Métodos Abstractos

- Un método es abstracto si no tiene una implementación.
- Una clase es abstracta si no permite tener instancias de ella.
  Ejemplo (JAVA):

```
abstract class Persistent {
    ...
   abstract state get-state();
   save() { medium.save(get-state()); }
    ...
    storage medium;
    ...
};

class Person extends Persistent {
    ...    state get-state() { return id+name+address; }
    ...
};
```

Delegación y Redirección

Efectos similares a la herencia.

- Se definen a nivel de objetos: si un objeto `o1` no comprende un mensaje `m`, éste se reenvía, en forma transparente, a un objeto `o2`.
- Se diferencian en como tratan el operador **self**.
- En redirección, `o1` y `o2` mantienen sus entidades separadas. En OZ es impl. por medio de **otherwise**.
- En delegación `o1` y `o2` referencian a la misma entidad. `{o2 setDelegate(o1)}`, `o1` se comporta como una superclase de `o2`.

Reflexión

Un sistema (u objeto) es reflectivo si éste puede inspeccionar parte de su estado de ejecución dinámicamente.

- Es **Introspectiva**: si solo permite leer su estado interno.
- Es **Intrusiva**: si permite modificar su estado.
- JAVA y Eiffel permiten reflexión introspectiva. Phyton y Ruby, permiten modificar componentes de objetos dinamicamente.

Meta clases

Las meteclases permiten hacer reflexión intrusiva.

- Ej. Smalltalk, Ruby o CLOS.
- JAVA, C++, Eiffel no disponen de metaclases (aunque el mecanismo es provisto por algunos métodos de clases específicas (Eiffel: INTERNALS y MEMORY, o algunos métodos de `Objects` de JAVA).

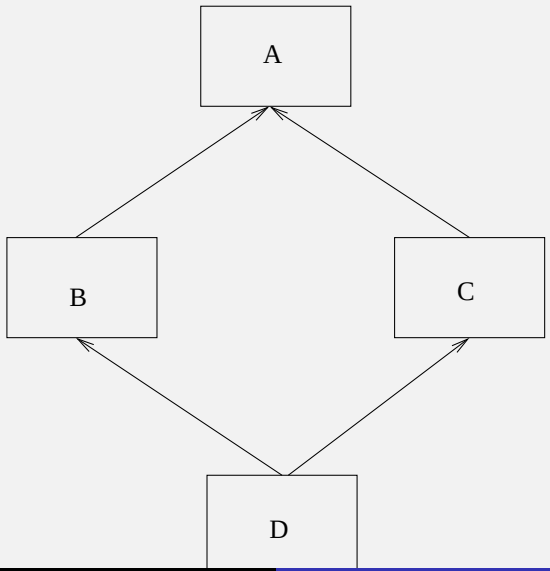
Constructores y Destructores

- **Constructor**: método que se invoca autom. cuando se crea un objeto.
- **Destructor**: método que se invoca explícitamente (C++) o implícitamente (por el recolector de basura, ej. JAVA, Eiffel, OZ) antes que el objeto se destruya.

Herencia Múltiple

Una clase hereda de 2 o mas clases.
Problemas:

- *Conflictos de nombres*: en el caso que las superclases tengan algún miembro con nombre común.
- *Repetición de atributos*: si dos (o más) clases, sean `B` y `C` heredan de una clase común (`A`) y una clase (`D`) hereda de `B` y `C`, se da el problema que `D` hereda por dos caminos diferentes de la clase `A` (Problema del Rombo).



Soluciones

Un lenguaje que permite Herencia Múltiple debe proveer mecanismo para solucionar los problemas.

- En C++: operador de alcance **::** para solucionar el primer problema. Para la repetición de atributos (**virtual**):

```
class A {...};
class B : virtual public A {...};
class C : virtual public A {...};
class D : public B, public C {...};
```

- En Eiffel se pueden renombrar o redefinir atributos y/o métodos.
- JAVA, solo permite herencia múltiple con las interfaces.

Generecidad

Mecanismo para definir algoritmos y tipos de datos en forma abstracta.

- El nombre de los tipos de datos de las abstracciones son parámetros de la definición.
- Se conoce como Polimorfismo estático. La herencia se conoce como Polimorfismo dinámico. Es posible combinarlas.
- Son esqueletos de funciones y tipos de datos (a diferencia del polimorfismo paramétrico).
- Sus instancias generan versiones específicas (a diferencia del polimorfismo paramétrico).

Genericidad en C++. Templates

Pueden aplicarse a funciones como clases y estructuras.

```
template <typename T>
T add(T const & v1, T const & v2)
{
   return v1 + v2;
}
```

```
template <typename T1, typename T2>
   struct pair {
       T1 first;
       T2 second;
   };
```

`add(5,x)`: crea una instancia de la función `int add(int const &v1, int const &v2) ...`

Generecidad en C++

C++ provee en su biblioteca estándar un conjunto de plantillas, tanto de funciones como estructuras de datos (*abstract containers*)

- Algoritmos abstractos, como `sort`, `accumulate`, `find`, etc.
- Ccontenedores abstractos se encuentran `vector`, `list`, `stack`, `map`, `set` y otros.

```
...
class case_less
{
    public:
        bool operator()(string const &left, string const &right) const
        {
            return strcasecmp(left.c_str(), right.c_str()) < 0;
        }
};

void print(string s)
{
    cout << s << endl;
}

int main(int argc, char *[]argv)
{
    vector<string> v(argv,argv + argc);
    sort(v.begin(), v.end(), case_less());
```

Generecidad en JAVA

Ejemplo de Genericidad Restringida.

```
 class Concesionario<T extends Coche> {
 private T[] coches =null;

 /**
^^I* Constructor del concesionario
^^I* @param coches array con los coches que contiene el concesionario*/

 public Concesionario(final T[] coches) {
^^Ithis .coches = coches;
 ^^I}
/**
^^I* Obtiene todos los coches del concesionario
^^I* @return array con todos los coches del concesionario*/

^^IT[] obtenerCoches() {
^^I^^Ireturn coches;
^^I}
}
```

Generecidad Restringida en Eiffel

```
Ejemplos:
^^IVECTOR [G - > NUMERIC]
^^IDICTIONARY [G, H - > COMPARABLE]
^^IORDENACION [G - > COMPARABLE]
```

- Sólo es posible instanciar G con descendientes de la clase R.
- Las operaciones permitidas sobre entidades de tipo G son aquellas permitidas sobre una entidad de tipo R.

Capítulo 10 - Concurrencia

Conceptos Básicos

- **Programa**.
- **Proceso**.Un programa en ejecución (instancia de un programa).
- **Sistemas Monotareas**. Solo puede ejecutarse una taera simultaneamente (MS-DOS).
- **Sistemas Multitareas**. Se pueden ejecutar varias tareas “simultáneamente” (varios procesos).

Definición y Características

Algunos programas, se pueden escribir más fácilmente como un conjunto de actividades que ejecutan independientemente. Tales programas se denominan concurrentes.

- Tales actividades pueden compartir recursos (variables, archivos, etc.).
- Necesidad de Mecanismos de sincronización.
- Los programas se escriben siguiendo alguna lógica de comportamiento de sus partes (cliente-servidor, productor-consumidor, etc.).

Concurrencia

Modelos de Ejecución

- *memoria compartida (shared memory)*: cada actividad o módulo concurrente del sistema accede a un área de memoria común. El mecanismo natural es el uso de variables compartidas. En este modelo, cada actividad se denomina comúnmente un *hilo (thread) de ejecución*.
- *pasaje de mensajes (message passing)*: las actividades tienen su propia área de memoria y se comunican por medio de *mensajes*, a través de *canales de comunicación (channels)*. En este modelo, es común que las actividades se denominen *procesos* o *tareas (tasks)*.

Concurrencia Declarativa

El mecanismo confía en el concepto de *variables data-flow (flujo de datos)*.

- Los resultados del programa serán los mismos, independientemente si se utiliza concurrencia o no.
- Lo nuevo es que un programa puede ser computado en forma incremental, a medida que arriban los datos de entrada.

Extension del Modelo Declarativo

Extendemos el Modelo Declarativo en 2 pasos:

- Primer Paso: Añadimos hilos y la instrucción `thread` $< d >$ `end`. Al modelo resultante lo llamamos el *modelo concurrente dirigido por los datos*.
- Segundo Paso: Se agregan disparadores por necesidad y la instrucción *{ByNeed P X}*. Al modelo resultante lo llamamos el *Modelo concurrente dirigido por la demanda* o *modelo concurrente perezoso*.

Concurrencia

Modelo Concurrente Dirigido por los Datos - Extensión del Modelo Declarativo

- **thread** <s> **end**.
- La máquina se extiende para tener varias pilas semánticas en lugar de una como en el modelo secuencial.

Semántica de los Threads

- La máquina soportará Múltiples pilas de ejecución (MST,σ), donde MST es un multiset.
- Al inicio el multiset tiene una sola pila. $(\{[< s >, \emptyset]\}, \emptyset)$
- En cada ejecución la máquina elige una pila (esto lo hace el *planificador (scheduler)*) y ejecuta la sentencia del tope.
- Termina si todas las pilas están vacias.
- Semantica de la sentencia:

$$(\{[\text{thread <s> end,E}]+ST'\} \biguplus MST', \sigma) \rightarrow (\{[(\text{<s>,E})]\} \biguplus ST \biguplus MST, \sigma)$$

Concurrencia

Ejemplo

```
local B in
    thread B = true end
    if B then {Browse 'se ligo B'} end
end
```

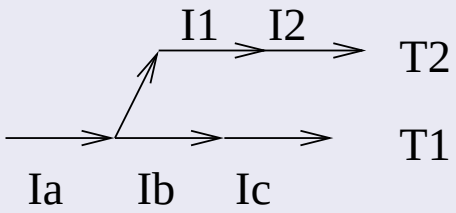
Orden de ejecución

Otra forma de ver la diferencia entre ejecución secuencial y concurrente es en términos de un orden entre todos los estados de ejecución de un programa cualquiera:

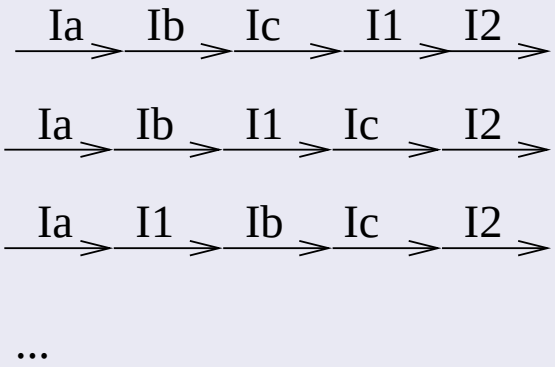
- En el modelo concurrente podemos obtener diferentes secuencias de estados de ejecución (dependiendo del thread elegido en cada paso de ejecución).
- Cada secuencia se denomina una *traza* de ejecución.
- Esta forma de ejecución secuencial de sentencias de diferentes threads en cada ciclo, se denomina *intercalación (interleaving)*.
- Dado un programa concurrente, sus pasos de ejecución forman un *orden causal*, el cual es un orden parcial (a diferencia de un programa secuencial que forma un orden total).

Definición:En un orden causal una sentencia s_i precede a otra s_j si ambas pertenecen al mismo thread y en él s_i se debe ejecutar antes que s_j.

Ejemplo



a) Orden causal



b) Posibles ordenes (interleaving)

Figure: Orden causal e intercalación (interleaving)

Que es Concurrencia Declarativa

El *modelo concurrente dirigido por los datos* es una forma de programación declarativa.
En el modelo concurrente declarativo pueden ocurrir dos cosas para todas las ejecuciones posibles de un programa:

- puede no terminar, o
- puede terminar parcialmente (computar valores parciales), terminan totalmente, y dichos valores son lógicamente equivalentes.

La concurrencia Declarativa produce *no-determinismo no observable*.

Utilizando un Programa Declarativo en un marco concurrente

Considere el ciclo `ForAll`.

```
proc {ForAll L P}
   case L of  nil then skip
   [] X|L2 then {P X} {ForAll L2 P} end
end

%Lo ejecutamos en un hilo:
declare L L1 L2 in
   thread {ForAll L Browse} end
   thread L = 1|L1 end
   thread L1 = 2|3|L2 end
   thread L2 = 4|nil end
```

Cual es la salida?.
Que diferencia hay con ForAll [1 2 3 4] Browse?.

Utilizando un Programa Declarativo en un marco concurrente

Versión Concurrente de `Map`.

```
fun {Map Xs F}
   case Xs of nil then nil
   [] X|Xr then thread {F X} end|{Map Xr F} end
end

declare F Xs Ys Zs
{Browse thread {Map Xs F} end}

....
Xs=1|2|Ys
fun {F X} X*X end
...
Ys=3|Zs
Zs=nil

Cual es la salida?
```

Utilizando un Programa Declarativo en un marco concurrente

```
fun {Generar N Limite}
   if N<Limite then
      N|{Generar N+1 Limite}
   else nil
   end
end

fun {Suma Xs A}
   case Xs of X|Xr then {Suma Xr A+X}
   [] nil then A
   end
end

local Xs S in
   thread Xs={Generar 0 150000} end    ^^I% Hilo productor
   thread S={Suma Xs 0} end^^I^^I% Hilo consumidor
   {Browse S}
end
% Modificar el programa para consumir los impares generados?
```

Control sobre los Procesos

Los threads se conocen como un mecanismo de *concurrencia cooperativa*.

- En cambio en un sist. Operativo los programas concurrentes compiten por los recursos.
- Un *thread* puede estar en estado: *runnable*, *blocked* o *termined*.
- Los lenguajes deben ofrecer primitivas de control sobre la ejecución de los threads.

Operación	**Descripción**
{Thread.this}	Retorna el nombre del thread corriente
{Thread.state T}	Retorna el estado del thread T
{Thread.suspend T}	Suspende al thread T
{Thread.resume T}	Activa al thread T
{Thread.preempt T}	Interrumpe (quita el control) al thread T
{Thread.terminate T}	Finaliza (inmediatamente) al thread T
{Thread.wait T}	Espera por la terminación del thread T

Corrutinas

Una corrutina es un *thread* no interrumpible.
Las corrutinas tienen dos operaciones fundamentales:

- **Spawn P**: crea una nueva corrutina con `P` como procedimiento principal de la corrutina y retorna un identificador de la corrutina creada.
- **Resume C**: transfiere el control a la corrutina `C`.

Barreras

Una barrera (barrier) es una operación de control de ejecución concurrente que permite establecer un punto de encuentro (punto de sincronización) de diferentes componentes concurrentes.

```
...
{Barrier [X1 X2 ... Xn]}
```

Modelo concurrente dirigido por la demanda (*modelo concurrente perezoso*)

Diferenciar entre *evaluación perezosa* y *ejecución perezosa*.

- *evaluación perezosa*: Una expresión se evalúa cuando su resultado es requerido en alguna parte del programa.

```
fun lazy {F1 X} 1+X*(3+X*(3+X)) end
fun lazy {F2 X} Y=X*X in Y*Y end
fun lazy {F3 X} (X+1)*(X+1) end
A={F1 10}
B={F2 20}
C={F3 30}
D=A+B
```

- La ejecución lazy es mas general que la *evaluación lazy*.
- El modelo de ejecución lazy extiende el modelo concurrente declarativo con un nuevo concepto: disparadores por necesidad (by need triggers).
- `{ByNeed P Y}`, tiene el mismo efecto que la sentencia {**thread** `{P Y}` **end**}, excepto para planificador, en el sentido que el procedimiento `{P Y}` será ejecutado (planificado) solamente si el valor `Y` es necesitado.

Semántica de la sentencia $(\{ByNeed\ < x >< y >\}, E)$

Se extiende la memoria para disponer de una *memoria de triggers*, τ.

1. si $E(< y >)$ no es determinado, crear el trigger $trig(E(< x >), E(< y >))$ a la memoria de triggers τ.
2. si $E(< y >)$ es determinado, crear un nuevo thread con la sentencia inicial (o cuerpo) $(\{< x >< y >\})$.

Un trigger se activa cuando existe un valor $trig(x, y)$ y se detecta una necesidad de y, es decir, existe un thread que está suspendido por y o se está ligando a y.
En la activación de un trigger $trig(x, y)$ se ejecutan las siguientes acciones:

1. Eliminar $trig(x, y)$ de la memoria de triggers τ.
2. Crear un nuevo thread cuya sentencia inicial sea $(\{< x >< y >\}, \{< x >\rightarrow x, < y >\rightarrow y\})$. O sea que produce la invocación al procedimiento ligado a x con el argumento ligado a y.

Ejemplos:

1.
```
{ByNeed proc {$ A} A=111*111 end Y}
{Browse Y}
```

2.
```
{ByNeed proc {$ A} A=111*111 end Y}
Z= Y + 1
{Browse Y}
```

Sincronización

Cuando un thread necesita un valor calculado por otro thread, el primero deberá esperar hasta que el resultado está disponible.

- *dataflow*: (con evaluación estricta). Las operaciones que requieren un valor, deberán esperar hasta que el valor está disponible.
- *bajo demanda*: (ejecución peresoza). El intento de ejecución de una operación, causa la evaluación de sus argumentos. El cálculo de los argumentos, causa un punto de sincronización entre las operaciones.

En un programa, la sincronización puede ser:

- *implícita*: los puntos de sincronización no son visibles en el texto del programa, como en el modelo de concurrencia declarativa descripto hasta ahora.
- *explicita*: los puntos de sincronización son visibles en el programa (cerrojos (locks) o monitores).

Concurrencia con estado compartido

La concurrencia en el modelo con estado, crea problemas para el desarrollo de programas.

- Los threads acceden (escriben y leen) a celdas compartidas. Ejemplo:

```
local X in
   {NewCell X 0}
   thread X := @X + 1 end
   thread X := @X + 1 end
   {Browse @X}
end
```

- Varios threads actualizando la misma variable produce *condiciones de carrera (race condition)*.
- La concurrencia con estado explícito produce *no determinismo observable*.
- En algunas situaciones no es posible escribir invariantes (el valor de una variable puede ser incierto).
- La progr. concurrente (con estado) se basa en la detección de las regiones críticas y establecer **sincronización** sobre las regiones.
- Es necesario introducir mecanismos de sincronización.

Primitivas de Sincronización

- **Cerrojos (locks)**: otorgan acceso exclusivo a un grupo de sentencias.
- **Semáforos**: un semáforo, inventados por Dijkstra, es una variable (de tipo entera) especial protegida (ADT), el cual puede ser manipulada por tres operaciones:
  - `init(s,n)`: inicializa el semáforo con el valor `n`.
  - `P(s)`: intenta decrementar en uno el valor del semánforo, si es que su valor es positivo. En otro caso el thread invocante deberá esperar (bloquearse) hasta que sea desbloqueado por otro thread.
  - `V(s)`: si hay algún proceso bloqueado por el semáforo, desbloquea uno y retorna, sino, incrementa en uno el valor del semáforo.
- **Monitores**: Garantizan la exclusión mutua. Generalmente incluyen operaciones para esperar por una cierta condición (`wait`) y notificación de condiciones (`signal`).

Concurrencia

Primitivas de Sincronización

- **Regiones críticas condicionales**: Un región crítica está protegida con un lock asociado a una condición. Un thread intentando ingresar a la región crica deberá esperar hasta que la condición sea verdadera.

  Generalmente tienen la forma de **`region ... when`** `<cond> <s>` **`end`**.

- **Transacciones**: una transacción es una secuencia de operaciones que pueden ejecutarse con éxito (`commit`) o ser abortadas. En el caso que se produzca una salida anormal (`abort`) el estado se retrotrae al estado previo del inicio de la transacción.

Primitivas de OZ

- `{NewLock L}`: crea un nuevo lock `L`.
- `{IsLock L}`: retorna **true** si `L` referencia un lock.
- **`lock`** `X then <s> end`: protege a la sentencia `<s>` con el lock `X`. Un thread ejecutando una sentencia protegida por un lock `X` impide que otro thread acceda (debe esperar) al cuerpo de la sentencia lock sobre la misma variable.

Concurrencia

Ejemplos usando cerrojos de OZ

```
 declare
   L = {NewLock}
   {NewCell X 0}
   thread lock L then X := @X + 1 end end
   thread lock L then X := @X + 1 end end
   {Browse @X}
end
Cual es la salida?
```

Ejemplo de Implem. en JAVA de la primitiva P() de Semáforos

```
 class Semaphore {
  public Semaphore(int init_val) { sem = init_val; }

  public synchronized void P()
     {
        while (sem == 0)
^^I  try {
                wait();
            } catch( InterruptedException e ) { ; }

^^Isem--;
     }
...
```

Parte de la Implem. de Problema de Productor-Consumidor usando semáforos

```
class SharedBuffer {
  // Constructor
  public SharedBuffer(int Capacity){
^^I   this.Capacity = Capacity;
^^I   mutex = new Semaphore(1);
^^I   full = new Semaphore(Capacity);
^^I   empty = new Semaphore(0);
^^I   System.out.println(Capacity);
^^I }

  public void append_elem(Object element){
^^I   full.P();^^I^^I^^I// block if buffer is full

^^I   mutex.P();^^I^^I// exclusive acess to buffer
^^I   impl_append(element);
^^I   mutex.V();

^^I   empty.V();^^I^^I// wakeup consumers
^^I}
```

Concurrencia

Consecuencia del uso de mecanismos de sincronización en el modelo concurrente con estado.

Deadlock

- Un estado en el cual dos o más threads o procesos podrían quedar esperándose mutuamente para acceder a los recursos.
- Este estado que imposibilita el progreso de componentes de un sistema concurrente se conoce como *deadlock* (abrazo mortal).

Ejemplo:

```
T1 = thread                    T2 = thread
...                            ...
lock X in                      lock Y in
   ... (1)                        ... (2)
   lock Y in                      lock X in
      ...                            ...
   end                            end
...                            ...
end                            end
```