Teoría de Lenguajes
Teoría de la Programación

Clase 2: Modelo computacional declarativo

Modelo computacional

Sistema formal que define cómo se ejecutan los cálculos. Se define en términos de los conceptos que incluye. Sintaxis y semántica

Un modelo computacional es una definición más precisa de un paradigma de programación.

¿Qué nos permite estudiar un modelo computacional?

Modelo computacional

- Correctitud
- Complejidad temporal
- Complejidad espacial

Modelo declarativo

Modelo declarativo

- Evaluar funciones sobre estructuras de datos(parciales)
  - Sin estado (Stateless)
  - Paralelizable
  - Deterministico
  - Sin efectos secundarios
  - Fácil de probar

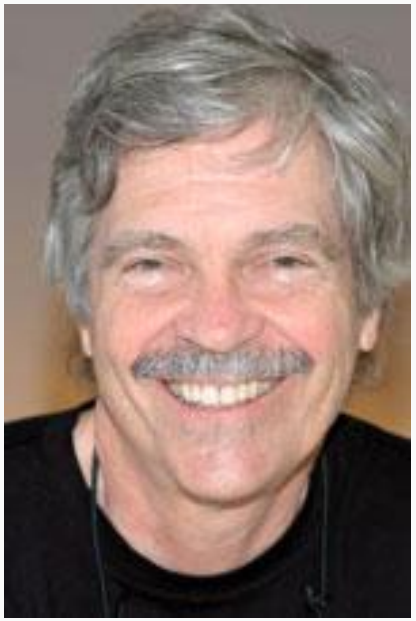
- Ideas principales del paradigma funcional y lógico

Modelo declarativo

Edsger Dijkstra (1930-2002)

“Object-oriented programming is an exceptionally bad idea which could only have originated in California”

Modelo declarativo



Alan Kay (1940)

Entre otras cosas, creador de Smalltalk

“You probably know that arrogance, in computer science, is measured in nanodijkstras”

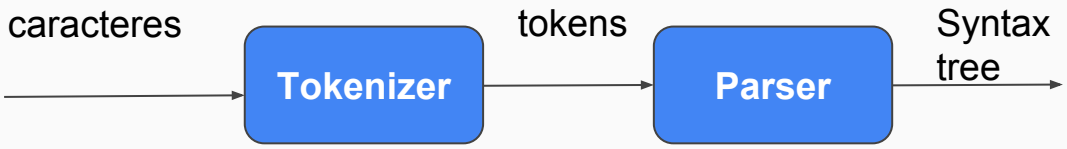
Modelo declarativo

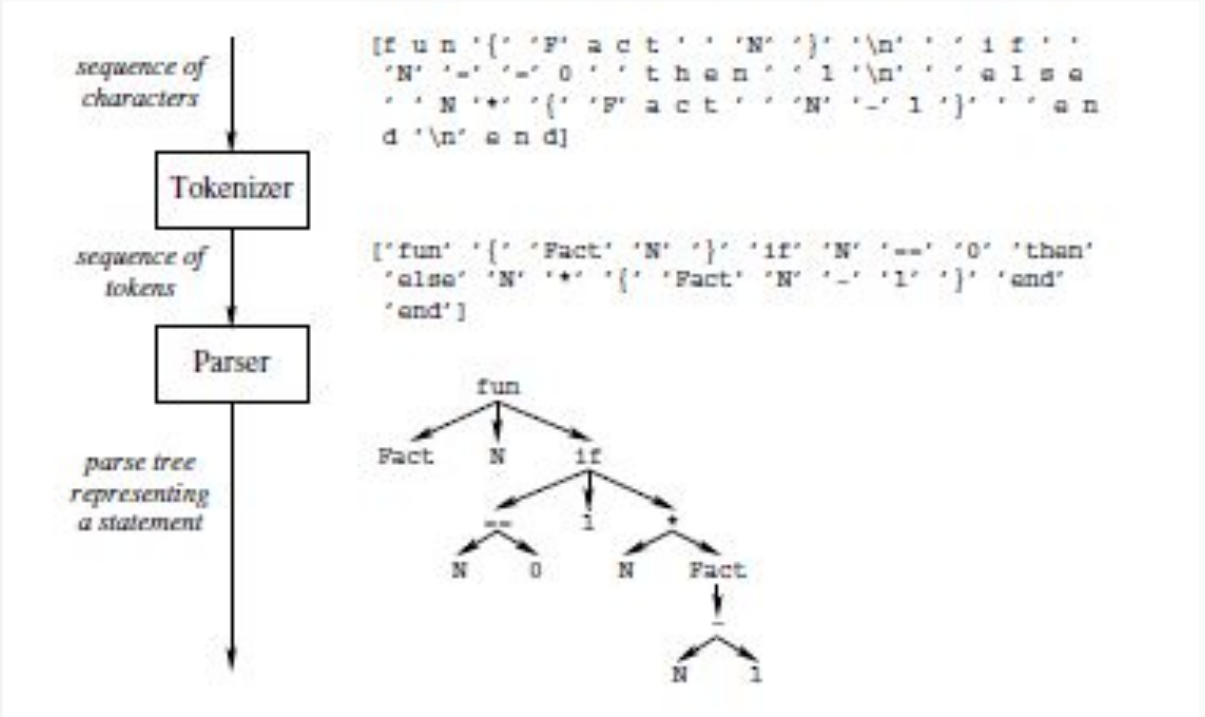
Una torta es 45 minutos de 200°C de calor aplicado a 200 mg de masa de torta final.
100 mg de masa de torta final es una mezcla de 99 mg de masa de torta etapa 2 y 1 mg de sal.
99 mg de masa de torta etapa 2 es una mezcla de 79 mg de harina y un huevo.

¿Cómo definimos un lenguaje?

Sintaxis

Qué es legal en un programa sin importar qué es lo que el programa esté haciendo





Sintaxis - EBNF

Extended Backus-Naur Form

<digit> ::== 0|1|2|3|4|5|6|7|8|9
<int> ::== <digit>{<digit>}

Semántica

Qué es lo que hace el programa al ejecutarse

Debe:

- Ser simple
- Permitir entender el programa en cuanto a correctitud y complejidad

Semántica - Kernel Language approach

- Set mínimo de instrucciones intuitivas
- Fácil de razonar, con semántica formal
- Se extiende a un lenguaje práctico a través de
  - Syntactic sugar
  - Linguistic abstraction

Lenguaje Kernel

Lenguaje Kernel - Valores

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

Lenguaje Kernel - Statements

⟨s⟩ ::=	
skip	Empty statement
| ⟨s⟩$_1$ ⟨s⟩$_2$	Statement sequence
| **local** ⟨x⟩ **in** ⟨s⟩ **end**	Variable creation
| ⟨x⟩$_1$=⟨x⟩$_2$	Variable-variable binding
| ⟨x⟩=⟨v⟩	Value creation
| **if** ⟨x⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Conditional
| **case** ⟨x⟩ **of** ⟨pattern⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Pattern matching
| {⟨x⟩ ⟨y⟩$_1$ ... ⟨y⟩$_n$}	Procedure application

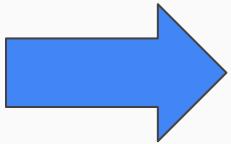
Expressions vs Statements

Una expresión es un valor. Es el resultado de una operación. Es algo que puede ser asignado a una variable

Un statement es una secuencia de operaciones que no devuelven valor

Múltiples declaraciones

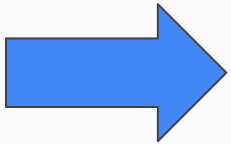
```
local A B in
   A = 4
   B = 5 + A
end
```



```
local A in
   local B in
      A = 4
      B = 5 + A
   end
end
```

Declaraciones sin asignación

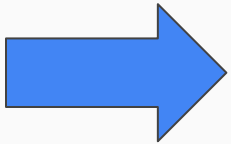
```
local X=<expression>
in
    <statement>
end
```



```
local X in
    X = <expression>
<statement>
end
```

Funciones a procedimientos

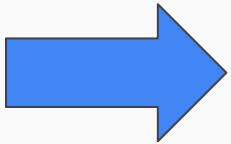
```
X = {F Y}
```



```
{F X Y}
```

Llamados anidados

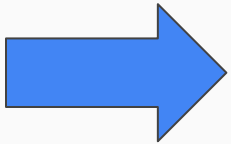
```
{P {F A X} Y}
```



```
local U in
    {F A X U}
    {P U Y}
end
```

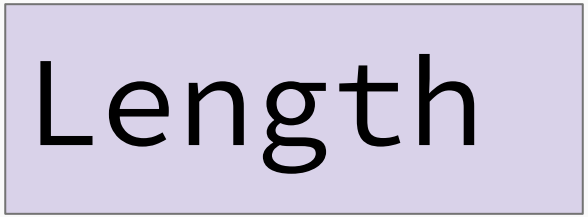
Condicionales booleanos

```
if X > Y then
    <statement>
end
```



```
local B in
    B = X > Y
    if B then
        <statement>
    end
end
```

Ejemplo



Scoping y procedures

Scoping - local scoping

```
local X in
        X=1
        local X in
            X=2
            {Browse X} % Muestra 2
        end
        {Browse X} % Muestra 1
    end
```

Scoping - static vs dynamic scoping

```
local P Q in
        proc {Q X} {Browse stat(X) end
        proc {P X} {Q X} end
        local Q in
            proc {Q X} {Browse dyn(X)} end
            {P ‘hola’}
        end
    end
```

Scoping - static vs dynamic scoping

```
local P Q in
        proc {Q X} {Browse stat(X) end
        proc {P X} {Q X} end
        local Q in
            proc {Q X} {Browse dyn(X)} end
            {P ‘hola’}
        end
    end
```

Abstraccion procedural

Cualquier statement puede convertirse en un procedimiento

En un statement un identificador es libre si no está definido en el mismo

Máquina abstracta

Single Assignment Store

σ

- Variables declarativas
- Value store
- Partial values
- Variables dataflow

Ej:

$\sigma = \{x_1=100, x_2=[1\ 2\ 3], x_3\}$

Entorno

E

- Identificadores

Notación:

<x> identificador de una variable

E(<x>) el valor en el store del identificador <x> según su entorno

Operaciones:

Adición: E’ = E + {<x>-> x}

Restricción: $E' = E|_{\{<x>,..<z>\}}$

STACK

ST

- Pila de semantic statements

Semantic statement es un par (<s>,E) siendo <s> un statement

Cómputo

- Estado de ejecución.
- Un cómputo define cómo se modifica el estado de ejecución de un programa.

$$(ST_0, \sigma_0) \rightarrow (ST_1, \sigma_1) \rightarrow (ST_2, \sigma_2) \rightarrow \ldots$$

Estados de ejecución

- Ejecutable
- Terminado
- Suspendido

Semántica

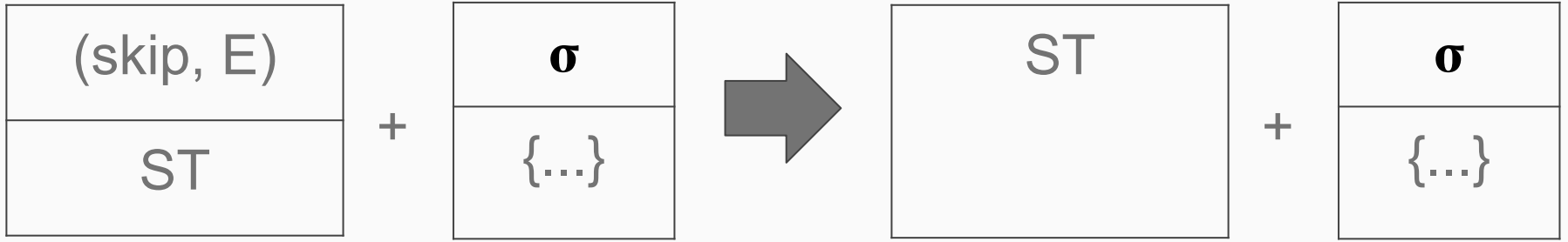
Statements

⟨s⟩ ::=	
skip	Empty statement
⟨s⟩$_1$ ⟨s⟩$_2$	Statement sequence
local ⟨x⟩ **in** ⟨s⟩ **end**	Variable creation
⟨x⟩$_1$=⟨x⟩$_2$	Variable-variable binding
⟨x⟩=⟨v⟩	Value creation
if ⟨x⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Conditional
case ⟨x⟩ **of** ⟨pattern⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Pattern matching
{ ⟨x⟩ ⟨y⟩$_1$ ... ⟨y⟩$_n$ }	Procedure application

Skip

En el tope del ST tenemos el siguiente semantic statement (skip, E)

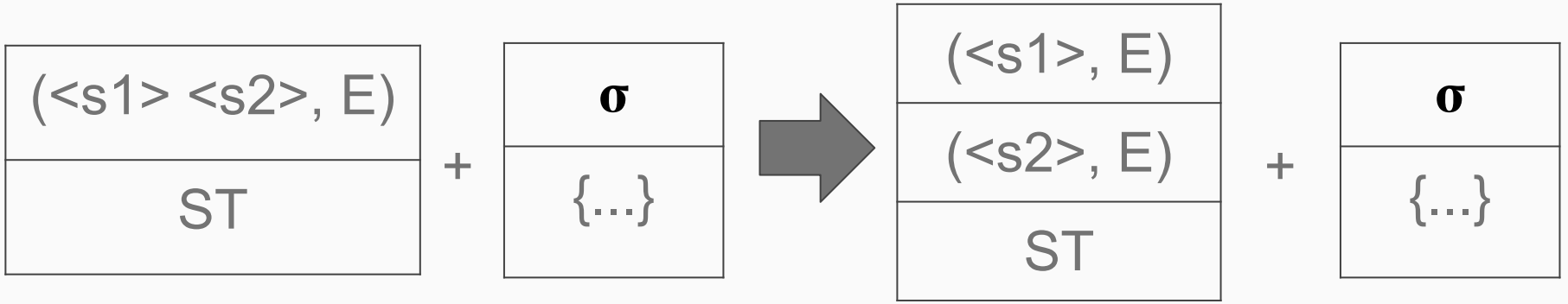
Se continua con el próximo paso



Composición secuencial

En el tope del ST tenemos el siguiente semantic statement (<s_1> <s_2>, E)

Se apila cada statement en el ST con el mismo entorno de la composición



Declaración de variable

En el tope del ST tenemos el siguiente semantic statement
(local <x> in <s> end , E)

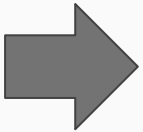
- Se crea la variable x en el store
- Se crea un ambiente E' = E + {<x> -> x}. Es decir un ambiente igual al anterior pero con el identificador <x> mapeando a la variable recién creada
- Se apila (<s>,E') al ST
- Se continua con la próxima ejecución

Declaración de variable

(local A in <s> end, E)
ST

+

σ
{...}



(<s>, E')
ST

+

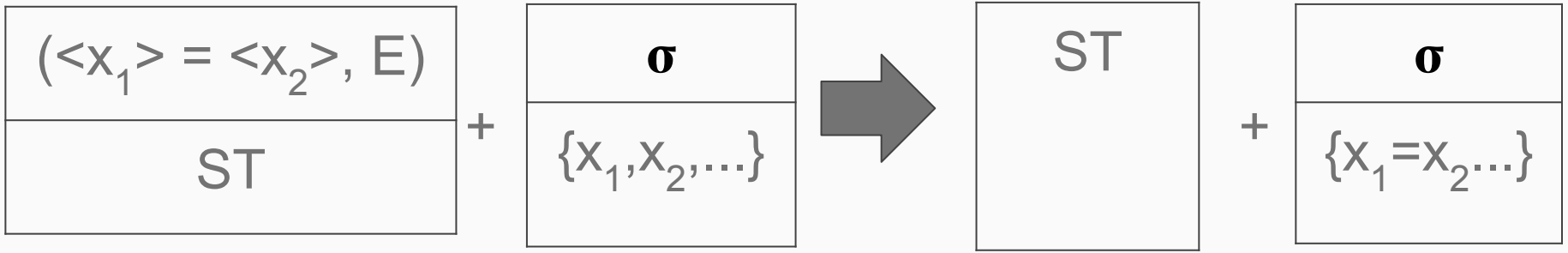
σ
{..., a_1}

- E' = E + {A->a_1}
- a_1 no ligada en σ

Igualdad variable - variable

En el tope del ST tenemos el siguiente semantic statement ($\langle x_1 \rangle = \langle x_2 \rangle$, E)

Se hace un bind de E($\langle x_1 \rangle$) con E($\langle x_2 \rangle$) en el store

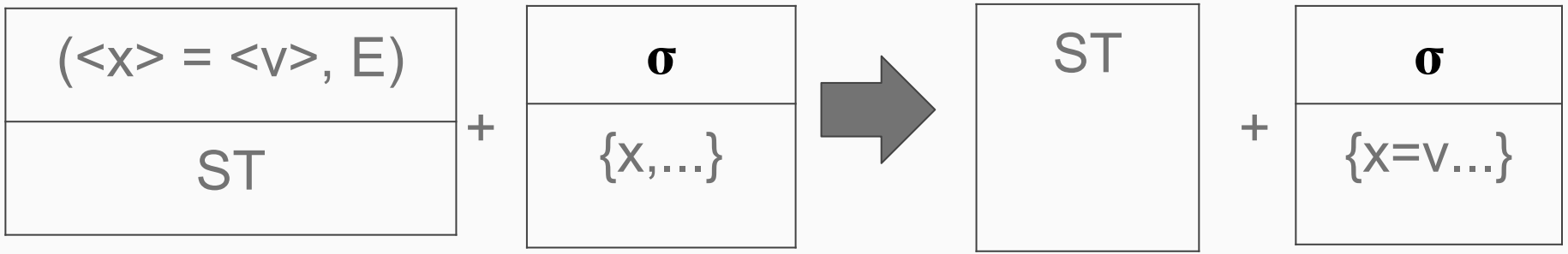


Igualdad variable - valor

En el tope del ST tenemos el siguiente semantic statement (<x> = <v>, E)

<v> es un record, un numero o un procedimiento

Se crea el valor y se liga a la variable <x> en el store



WHAAAT?

EJEMPLO

```
local X Y Z in
    X = 10
    Y = 40
    Z = X
    local X A in
        X = 30
        A = persona(nombre:'Lean' edad:X)
    end
end
```

EJEMPLO || - Procedures

```
local X Y Z P in
    X = 10
    Y = 40
    proc {P A B C}
        C = A + B
    end
    {P X Y Z}
    {Browse Z}
end
```

```
local X Y Z P in
    X = 10
    Y = 40
    proc {P A C}
        C = A + Y
    end
    {P X Z}
    {Browse Z}
end
```

Procedure values (closures)

tenemos el siguiente semantic statement (<x> = <v>, E) siendo <v> un procedure value

<v> = proc {$ $\langle y_1 \rangle$ ..$\langle y_N \rangle$} $\langle s_P \rangle$ end

Analizar los identificadores libres de $\langle s_p \rangle$

1. Parametros formales
2. Referencias externas

Procedure values (closures)

Se guarda en el store el par (proc {$ <y_1> ..<y_N>} <s_P> end, CE)

Siendo CE = $E|_{\{<z1>,...,<zk>\}}$

Con {<z1>,...,<zk>} referencias externas se <s_p> en E

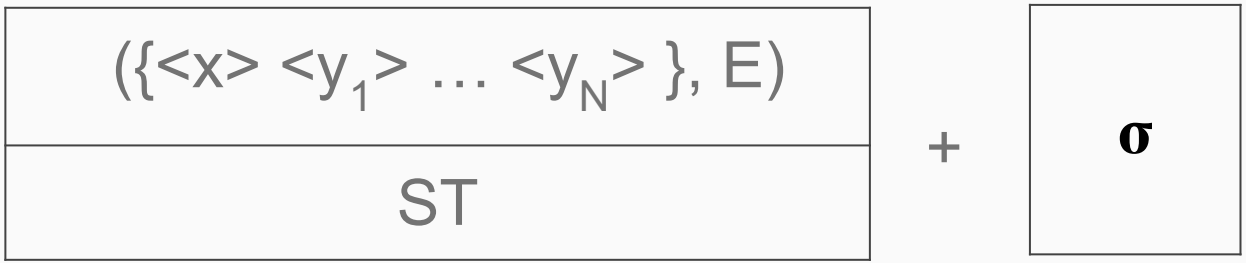
Procedure application

En el tope del ST tenemos ({<x> <y_1> … <y_N> }, E)

Es **suspendible**. Tiene un trigger de activación:
    E(<x>) tiene que estar determinado. Sino suspende

Si E(<x>) está determinado.
    Tiene que ser un procedure value con aridad igual a N.
Sino lanza error

Procedure application



σ = {x=(proc{$ <z1>...<zN>} <sP> end, CE), y1,...,yN,...}
E = { <x> -> x, <y1>-> y1, … , <yN> -> yN,...}

Se apila al ST (<sP>, EP) con
EP = CE + { <z1> -> E(<y1>), …, <zN> ->E(<yN>)}

Volvamos al EJEMPLO || - Procedures

```
local X Y Z P in
    X = 10
    Y = 40
    proc {P A B C}
        C = A + B
    end
    {P X Y Z}
    {Browse Z}
end
```

```
local X Y Z P in
    X = 10
    Y = 40
    proc {P A C}
        C = A + Y
    end
    {P X Z}
    {Browse Z}
end
```

Condicional

En el tope del ST tenemos (if <x> then <s_1> else <s_2> end, E)

Es **suspendible**. Tiene un trigger de activación:
E(<x>) tiene que estar determinado. Sino suspende

Si E(<x>) está determinado.
Tiene que ser un boolean, sino lanza error
Si es **true** apila (<s_1>,E), si es **false** apila (<s_2>,E) al ST

Ejemplo III

```
local Max C in
    fun {Max X Y}
        if X >=Y then X else Y end
    end
    C = {Max 3 5}
end
```

Ejemplo IV

```
local LowerBound Y C in
    Y = 5
    proc {LowerBound X Z}
        if X>=Y then Z=X else Z=Y end
    end
    {LowerBound 3 C}
end
```

Ejemplo V

```
fun {Fact1 X}
   if X ==0 then 1 else X * {Fact1 X-1} end
end
```

```
fun {Fact2 X Acum}
   if X==0 then Acum
   else
     {Fact2 X-1 Acum*X}
   end
end
```

Last call optimization

Tail recursion

Optimiza el uso de memoria

El stack no crece dependiendo de los elementos a procesar

Manejo de memoria

Valor alcanzable (**reachable**)

Un valor es alcanzable es referenciado en el ST o es referenciado por algún otro valor que sea alcanzable

Memoria activa (**active memory**)

La memoria activa se compone por el ST y por los valores alcanzables del store

Manejo de memoria

El manejo de memoria puede ser manual o a traves de un **garbage collector**

Problemas:

- Referencias colgadas (dangling references)
- Perdidas de memoria (memory leaks)

Más semántica

Pattern matching

Tenemos el siguiente semantic statement
(case <x> of <l>(<f_1>:<x_1> … <f_n>:<x_n>) then <s_1> else <s_2> end, E)

Se activa si E(<x>) está definido, sino **suspende**
Si E(<x>) = record con label <l> y aridad [<f_1> … <f>].
Hace push al ST de:
(local <x_1>=<x>.<f_1> … <x_n>=<x>.<f_n> in <s_1> end, E)
Sino, hace push de **(<s_2>, E)** al ST

Excepciones - try

Tenemos el siguiente semantic statement
(try <s_1> catch <x> then <s_2> end, E)

1. Se apila el st (catch <x> then <s_2> end, E) en ST
2. Se apila (<s_1>, E) en ST

Excepciones - raise

Tenemos el siguiente semantic statement
(raise <x> end, E) en donde el raise puede ser implícito o explícito

Se empiezan a descartar elementos del ST hasta encontrar un catch. Si no hay catch, se finaliza la ejecución con “**Uncaught execption**”

Si hay un catch apilado. Supongamos **(catch <y> then <s> end, E_c)**
Se apila **(<s>, E_c + {<y> ->E(<x>)})**

Bibliografía

- **Concepts, Techniques, and Models of Computer Programming - Capítulo 2**, Peter Van Roy and Seif Haridi
- **Extras:**
  - **A practical introduction to functional programming (with Python)**

    https://maryrosecook.com/blog/post/a-practical-introduction-to-functional-programming

  - **Tail call** https://en.wikipedia.org/wiki/Tail_call
  - **Garbage Collection: Algorithms for Automatic Dynamic Memory Management.** Wiley (1996). Richard Jones & Rafael D. Lins
  - **Abstract syntax tree** https://en.wikipedia.org/wiki/Abstract_syntax_tree