MACROS

- Problema: lo que queremos es
  - Más *comodidad* para escribir los programas
  - *Evitar* tener que *repetir* código
- Restricción: lo que no podemos permitirnos es
  - Introducir **nuevas primitivas** en el lenguaje
  - Permitir **objetos de usuario** en el lenguaje
- Solución: lo que haremos es
  - Definir un **código intermedio** más cercano al usuario: los MACROPROGRAMAS
  - Los macroprogramas deben funcionar como *abreviaturas* de los programas-while
  - Los macroprogramas deben ser *equivalentes* a los programas-while
  - Los macroprogramas deben constituir un *lenguaje abierto*

EXPANSIÓN DE MACROVARIABLES (EJEMPLO DE LA INVERSA X^R)

- MACROPROGRAMA

```
AUX := cons_a1(X1);
AUX := cdr(AUX);
while nonem?(AUX) loop
    if car_a1?(AUX) then
        X0 := cons_a1(X0);
    end if;
    if car_a2?(AUX) then
        X0 := cons_a2(X0);
    end if;
    •••
    if car_an?(AUX) then
        X0 := cons_an(X0);
    end if;
    AUX := cdr(AUX);
end loop;
```

NO

es un programa-while

- EXPANSIÓN

```
X2 := cons_a1(X1);
X2 := cdr(X2);
while nonem?(X2) loop
    if car_a1?(X2) then
        X0 := cons_a1(X0);
    end if;
    if car_a2?(X2) then
        X0 := cons_a2(X0);
    end if;
    •••
    if car_an?(X2) then
        X0 := cons_an(X0);
    end if;
    X2 := cdr(X2);
end loop;
```

SI

es un programa-while

EXPANSIÓN DE MACROEXPRESIONES: EJEMPLO DE LA CONCATENACIÓN X•Y

X0 := X2;

AUX:= $X1^R$;

while nonem?(AUX) loop

    if car_{a_1}?(AUX) then X0 := $cons_{a_1}$(X0); end if;

    if car_{a_2}?(AUX) then X0 := $cons_{a_2}$(X0); end if;

    •••

    if car_{a_n}?(AUX) then X0 := $cons_{a_n}$(X0); end if;

    AUX := cdr(AUX);

end loop;

- Tenemos dos macroexpresiones a expandir

- En la expansión de cada una hay tres fases:
  - Carga de los argumentos
  - Ejecución de la subrutina
  - Descarga del resultado

Z1 := $cons_{a_1}$(X2); Z1 := cdr(Z1); -- Carga de y desde X2 Z0 := $cons_{a_1}$(Z1); Z0 := cdr(Z0); -- Cálculo de id(y) X0 := $cons_{a_1}$(Z0); X0 := cdr(X0); -- Descarga de id(y) en X0	X0 := X2;

Y1 := $cons_{a_1}$(X1); Y1 := cdr(Y1); -- Carga de x desde X1
Y2 := $cons_{a_1}$(Y1); Y2 := cdr(Y2); -- Empieza cálculo de x^R
while nonem?(Y2) loop
if car_{a_1}?(Y2) then Y0 := $cons_{a_1}$(Y0); end if;
if car_{a_2}?(Y2) then Y0 := $cons_{a_2}$(Y0); end if;
•••
if car_{a_n}?(Y2) then Y0 := $cons_{a_n}$(Y0); end if;
Y2 := cdr(Y2);
end loop; -- Termina el cálculo de x^R
AUX:= $cons_{a_1}$(Y0); AUX:= cdr(AUX); -- Descarga de x^R en AUX
	AUX := $X1^R$;

```
while nonem?(AUX) loop
    if car_a1?(AUX) then X0 := cons_a1(X0); end if;
    if car_a2?(AUX) then X0 := cons_a2(X0); end if;
    •••
    if car_an?(AUX) then X0 := cons_an(X0); end if;
    AUX := cdr(AUX);
end loop;
```

LEY DE LAS CONSTANTES

Toda FUNCIÓN CONSTANTE
es while-computable

- Ejemplo:

$$f(x) = abcbb$$

- Dada cualquier palabra $\mathbf{w} \in \Sigma^*$, la función constante:

$$\mathbf{K_w}: \Sigma^* \longrightarrow \Sigma^*$$

$$\mathbf{K_w}(\mathbf{x}) = \mathbf{w}$$

es while-computable

- Demostración: si $\mathbf{w} = \mathbf{s_1 s_2 s_3 \ldots s_n}$

X0 := ε;

X0 := $cons_{s_n}$(X0);

•••

X0 := $cons_{s_1}$(X0);

- Ejemplo de macro-expresión que nos permite:

ABC := 'abc';

LEY DE LA COMPOSICIÓN LOCAL

La COMPOSICIÓN LOCAL de funciones while-computables es while-computable

- Ejemplos:

$$f(x,y,z) = x \bullet y \bullet z$$

$$snoc_a(x) = (cons_a(x^R))^R$$

$$rdc(x) = (cdr(x^R))^R$$

- Si son while-computables las funciones:

$$\psi: \ \Sigma^{*\mathbf{j}} \longrightarrow \Sigma^*$$

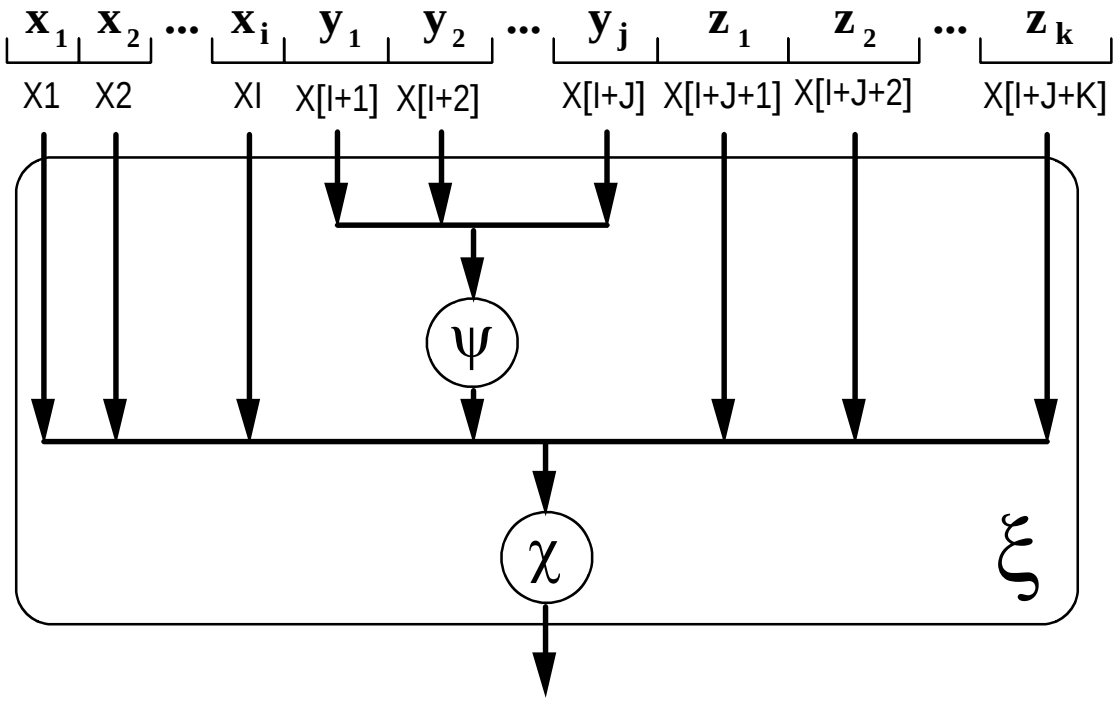
$$\chi: \ \Sigma^{*\mathbf{i}} \times \Sigma^* \times \Sigma^{*\mathbf{k}} \longrightarrow \Sigma^*$$

entonces también lo es la COMPOSICIÓN LOCAL de χ con ψ sobre la componente $\mathbf{i}+1$

$$\xi: \ \Sigma^{*\mathbf{i}} \times \Sigma^{*\mathbf{j}} \times \Sigma^{*\mathbf{k}} \longrightarrow \Sigma^*$$

$$\xi(\overline{\mathbf{x}}, \overline{\mathbf{y}}, \overline{\mathbf{z}}) \cong \chi(\overline{\mathbf{x}}, \psi(\overline{\mathbf{y}}), \overline{\mathbf{z}})$$

donde $\overline{\mathbf{x}} \in \Sigma^{*\mathbf{i}}$, $\overline{\mathbf{y}} \in \Sigma^{*\mathbf{j}}$, $\overline{\mathbf{z}} \in \Sigma^{*\mathbf{k}}$



- Demostración:

  X[I+J+K+1] := ψ(X[I+1], ..., X[I+J]);

  X0 := χ(X1, ..., XI, X[I+J+K+1], X[I+J+1],..., X[I+J+K]);

- Ejemplos de macro-expresiones que nos permite:

  X0 := (X0 & AUX)^R;

  donde ψ(**y₁,y₂**)= **y₁•y₂** χ(**u**)= **u**^{R}

  CASO := 'abba' & PAL;

  donde ψ(**y**)='abba' χ(**u,z**)=**u•z**

  A := B & D & A;

  donde ψ(**y₁,y₂**)= **y₁•y₂** χ(**x,u**)=**x•u**

PARTICULARIZACIONES

- Dada una función ψ: $\Sigma^{*\mathbf{k}+1} \longrightarrow \Sigma^*$, una PARTICULARIZACIÓN de ψ es otra función de menos argumentos que se define a partir ψ, fijando alguno de los que tiene

- Ejemplos:

$$f(x) = x \bullet \text{'abab'}$$

$$g(x) = x \bullet x$$

  son particularizaciones de la concatenación **x•y**

- Para obtener una particularización de ψ fijando el **i**-ésimo argumento se puede hacer de dos formas:

  * Fijando dicho argumento mediante una constante o valor fijo $\mathbf{w} \in \Sigma^*$

$$\chi(\overline{x}, \overline{y}) \cong \psi(\overline{x}, \mathbf{w}, \overline{y})$$

  * Fijando dicho argumento al igualarlo siempre a otro:

$$\xi(\overline{x}, \overline{y}) \cong \psi(\overline{x}, \mathbf{x_k}, \overline{y})$$

LEY DE LAS PARTICULARIZACIONES

Las PARTICULARIZACIONES de una función while-computable son while-computables

- Si $\mathbf{w} \in \Sigma^*$ es una palabra cualquiera, y la función:

$$\psi: \Sigma^{*\mathbf{i}} \times \Sigma^* \times \Sigma^{*\mathbf{j}} \longrightarrow \Sigma^*$$

es while-computable, entonces también lo son la PARTICULARIZACIÓN de ψ sobre la componente $\mathbf{i}+1$ mediante el valor $\mathbf{w}$:

$$\chi: \Sigma^{*\mathbf{i}} \times \Sigma^{*\mathbf{j}} \longrightarrow \Sigma^*$$

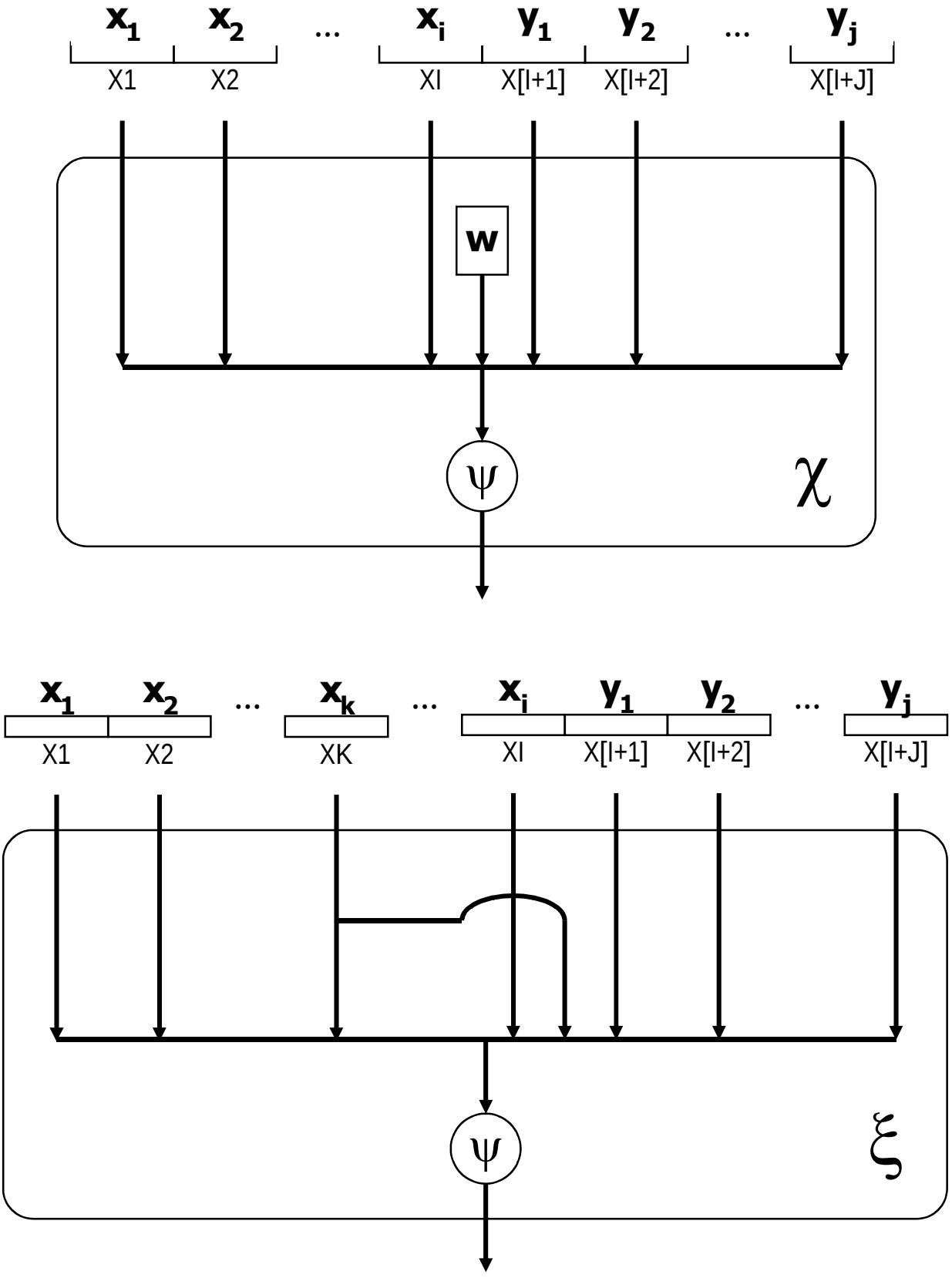
$$\chi(\overline{x}, \overline{y}) \cong \psi(\overline{x}, \mathbf{w}, \overline{y})$$

y la PARTICULARIZACIÓN de ψ sobre la componente $\mathbf{i}+1$ mediante la componente $\mathbf{k}$:

$$\xi: \Sigma^{*\mathbf{i}} \times \Sigma^{*\mathbf{j}} \longrightarrow \Sigma^*$$

$$\xi(\overline{x}, \overline{y}) \cong \psi(\overline{x}, \mathbf{x_k}, \overline{y})$$

donde $\overline{\mathbf{x}} \in \Sigma^{*\mathbf{i}}$, $\overline{\mathbf{y}} \in \Sigma^{*\mathbf{j}}$, $\mathbf{k} \leq \mathbf{i}$



- Demostración a partir de la Ley de Composición Local

LA FUNCIÓN *SIG*

- Nos permite establecer un orden entre palabras
- Se define inductivamente:

$$sig(\varepsilon) = a_1$$

$$sig(w \bullet a_i) = \begin{cases} w \bullet a_{i+1} & i < n \\ sig(w) \bullet a_1 & i = n \end{cases}$$

	$\Sigma=\{a\}$	$\Sigma=\{a,b\}$	$\Sigma=\{a,b,c\}$	$\Sigma=\{0,1\}$
w_0	ε	ε	ε	ε
w_1	**a**	**a**	**a**	**0**
w_2	**aa**	**b**	**b**	**1**
w_3	**aaa**	**aa**	**c**	**00**
w_4	**aaaa**	**ab**	**aa**	**01**
w_5	**aaaaa**	**ba**	**ab**	**10**
w_6	**aaaaaa**	**bb**	**ac**	**11**
w_7	**aaaaaaa**	**aaa**	**ba**	**000**
w_8	**aaaaaaaa**	**aab**	**bb**	**001**
w_9	**aaaaaaaaa**	**aba**	**bc**	**010**
w_{10}	**aaaaaaaaaa**	**abb**	**ca**	**011**

PROGRAMA PARA *SIG* CON Σ = {a, b, c}

```
X0 := ε;   AUX := X1^R;   SEGUIR := 'a';
-- sig(AUX^R)•X0=sig(X1) ∧ (SEGUIR=ε → ¬car_c?(AUX))
while nonem? (SEGUIR) loop
      SEGUIR := ε;
      if car_c? (AUX) then  SEGUIR := 'a';
            AUX:= cdr(AUX);
            X0 := cons_a(X0);
      end if;
end loop;
SOLO_C := 'a';
-- sig(AUX^R)•X0=sig(X1) ∧ ¬car_c?(AUX)
while nonem? (AUX) loop
      SOLO_C := ε;
      if car_a? (AUX) then   X0 := cons_b(X0);   end if;
      if car_b? (AUX) then   X0 := cons_c(X0);   end if;
      X0 := (cdr(AUX))^R & X0;   AUX:= ε;
end loop;
if car_a? (SOLO_C) then
-- sig(ε)•X0=sig(X1)
      X0 := cons_a(X0);
end if;
```

EXPANSIÓN DE MACROCONDICIONES: EJEMPLO DEL PREFIJO (PREFIJO?(X,Y))

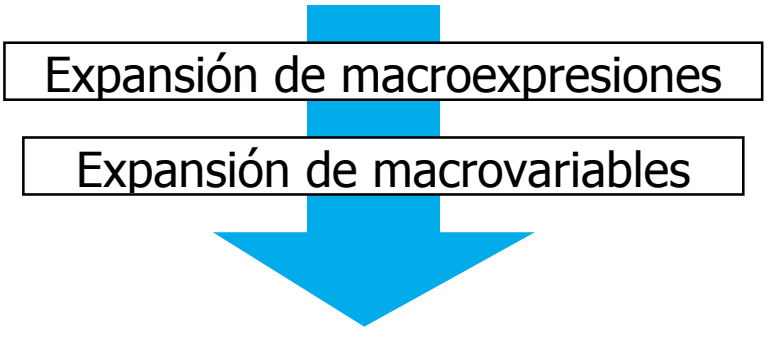
```
X0 := 'a1';   AUX1 := X1;   AUX2 := X2;
while   igual_comienzo? (AUX1, AUX2)   loop
        AUX1 := cdr(AUX1);   AUX2 := cdr(AUX2);
end loop;
if   nonem? (AUX1)   then   X0 := ε;   end if;
```

- Expansión de la macrocondición:

X0 := 'a$_1$'; AUX1 := X1; AUX2 := X2;

Z := $C_{igual\_comienzo?}$ (AUX1, AUX2);
while nonem? (Z) loop
AUX1 := cdr(AUX1); AUX2 := cdr(AUX2);
Z := $C_{igual\_comienzo?}$ (AUX1, AUX2);
end loop;

if nonem? (AUX1) then X0 := ε; end if;



PROGRAMA-WHILE

COMPOSICIÓN LOCAL DE PREDICADOS

La COMPOSICIÓN LOCAL de un predicado while-decidible con una función while-computable y total es while-decidible

$$P(x) = car_a?(cdr(cdr(x)))$$

$$Q(x,y) = nonem?(x \bullet y)$$

$$R(x,y) = prefijo?(x^R, y)$$

- Si la siguiente función *total* es while-computable y el siguiente predicado es while-decidible:

$$\mathbf{f}: \Sigma^{*\mathbf{j}} \longrightarrow \Sigma^*$$

$$\mathbf{P}: \Sigma^{*\mathbf{i}} \times \Sigma^* \times \Sigma^{*\mathbf{k}} \longrightarrow B$$

entonces también es while-decidible la COMPOSICIÓN LOCAL de **P** con **f** sobre la componente **i**+1

$$\mathbf{Q}: \Sigma^{*\mathbf{i}} \times \Sigma^{*\mathbf{j}} \times \Sigma^{*\mathbf{k}} \longrightarrow B$$

$$\mathbf{Q}(\overline{\mathbf{x}}, \overline{\mathbf{y}}, \overline{\mathbf{z}}) \Leftrightarrow \mathbf{P}(\overline{\mathbf{x}}, \mathbf{f}(\overline{\mathbf{y}}), \overline{\mathbf{z}})$$

donde $\overline{\mathbf{x}} \in \Sigma^{*\mathbf{i}}$, $\overline{\mathbf{y}} \in \Sigma^{*\mathbf{j}}$, $\overline{\mathbf{z}} \in \Sigma^{*\mathbf{k}}$

PARTICULARIZACIÓN DE PREDICADOS

Las PARTICULARIZACIONES de un predicado while-decidible son while-decidibles

- Si el predicado:

$$\mathbf{P}:\ \Sigma^{*\mathbf{i}} \times \Sigma^{*} \times \Sigma^{*\mathbf{j}} \longrightarrow B$$

es while-decidible, entonces también lo son las PARTICULARIZACIONES de **P** sobre la componente **i**+1, mediante el valor **w** o mediante la componente **k**:

$$\mathbf{Q}(\overline{x}, \overline{y}) \Leftrightarrow \mathbf{P}(\overline{x}, \mathbf{w}, \overline{y})$$

$$\mathbf{R}(\overline{x}, \overline{y}) \Leftrightarrow \mathbf{P}(\overline{x}, \mathbf{x_k}, \overline{y})$$

donde $\overline{\mathbf{x}} \in \Sigma^{*\mathbf{i}}$, $\overline{\mathbf{y}} \in \Sigma^{*\mathbf{j}}$, $\mathbf{k} \leq \mathbf{i}$

- Ejemplo de macro-expresiones que nos permiten:

while nonem?(cdr(cdr(PAL))) loop •••

if car_a?(X1&AUX^R) then •••

if prefijo?(PAL, X1&X1) loop •••

LEY DE LOS OPERADORES LÓGICOS

Al aplicar OPERADORES LÓGICOS (conjunción, disyunción o negación) sobre predicados while-decidibles produce predicados while-decidibles

$$P(x) \Leftrightarrow x = \varepsilon \Leftrightarrow \neg nonem?(x)$$

$$Q(x, y) \Leftrightarrow car_a?(x) \wedge car_a?(y)$$

- Si son while- decidibles los predicados:

$$\mathbf{P}: \Sigma^{*\mathbf{i}} \rightarrow B \quad \mathbf{Q}: \Sigma^{*\mathbf{j}} \rightarrow B$$

entonces también lo son los predicados:

$$\mathbf{R}(\overline{x}, \overline{y}) \Leftrightarrow \mathbf{P}(\overline{x}) \wedge \mathbf{Q}(\overline{y})$$

$$\mathbf{S}(\overline{x}, \overline{y}) \Leftrightarrow \mathbf{P}(\overline{x}) \vee \mathbf{Q}(\overline{y})$$

$$\mathbf{T}(\overline{x}) \Leftrightarrow \neg \mathbf{P}(\overline{x})$$

donde $\overline{\mathbf{x}} \in \Sigma^{*\mathbf{i}}$, $\overline{\mathbf{y}} \in \Sigma^{*\mathbf{j}}$

- CASO 1 (conjunción):

```
X0 := ε;
if P(X1, ..., XJ) then
    if Q(X[J+1], ..., X[J+K]) then
        X0 := 'a1';
    end if;
end if;
```

- CASO 2 (disyunción):

```
X0 := ε;
if P(X1, ..., XJ) then X0 := 'a1'; end if;
if Q(X[J+1], ..., X[J+K]) then X0 := 'a1'; end if;
```

- CASO 3 (negación):

```
X0 := 'a1';
if P(X1, ..., XJ) then X0 := ε; end if;
```

- Ejemplos de macros que nos permite:

```
while nonem?(AUX1) or nonem?(AUX2) loop •••

if cara?(PAL) and not prefijo?(PAL, X1) then •••
```

COMPARADORES Y WHILE-DECIDIBILIDAD

- El predicado .≤. : $\Sigma^* \times \Sigma^* \longrightarrow B$

```
AUX1 := X1;   AUX2 := X2;
-- X1≤AUX1 ∧ X2≤AUX2 ∧ (AUX1≤X2 ∨ AUX2≤X1)
while  AUX1 /= X2  and  AUX2 /= X1  loop
            AUX1 := sig(AUX1);   AUX2 := sig(AUX2);
end loop;
if  AUX1 = X2  then  X0 := 'a1';
else  X0 := ε;
end if;
```

- De aquí se derivan el resto de comparadores (mediante la utilización de operadores lógicos)

$$\mathbf{x} > \mathbf{y} \Leftrightarrow \neg \mathbf{x} \leq \mathbf{y}$$

$$\mathbf{x} \neq \mathbf{y} \Leftrightarrow \neg \mathbf{x} = \mathbf{y}$$

$$\mathbf{x} \geq \mathbf{y} \Leftrightarrow \mathbf{x} > \mathbf{y} \vee \mathbf{x} = \mathbf{y}$$

$$\mathbf{x} < \mathbf{y} \Leftrightarrow \mathbf{x} \leq \mathbf{y} \wedge \mathbf{x} \neq \mathbf{y}$$

LEY DE LA DEFINICION POR CASOS

Las funciones DEFINIDAS POR CASOS while-computables basados en predicados while-decidibles son while-computables

- Si son while-computables las funciones:

$$\psi : \Sigma^{*k} \longrightarrow \Sigma^*$$

$$\chi : \Sigma^{*k} \longrightarrow \Sigma^*$$

y es while-decidible el predicado

$$\mathbf{P} : \Sigma^{*k} \longrightarrow B$$

entonces también es while-computable la FUNCIÓN DEFINIDA POR CASOS basada en ψ, χ y **P**:

$$\xi(\overline{\mathbf{x}}) \cong \begin{cases} \psi(\overline{\mathbf{x}}) & \mathbf{P}(\overline{\mathbf{x}}) \\ \chi(\overline{\mathbf{x}}) & \text{c.c.} \end{cases}$$

donde $\overline{\mathbf{x}} \in \Sigma^{*\mathbf{k}}$

- Demostración:

  if P(X1, ..., XK) then X0 := ψ(X1, ..., XK);
  else X0 := χ(X1, ..., XK); end if;

- La ley se extiende de forma natural para la definición por casos de **predicados** while-decidibles

- También se extiende (por aplicación sucesiva) al caso en el que tenemos **n** funciones while-computables:

$$\psi_1, \ldots, \psi_n : \Sigma^{*k} \longrightarrow \Sigma^*$$

y **n**-1 predicados while-decidibles e incompatibles:

$$\mathbf{P_1}, \ldots, \mathbf{P_{n-1}} : \Sigma^{*k} \longrightarrow B$$

$$\mathbf{i \neq j} \wedge \mathbf{P_i}(\overline{\mathbf{x}}) \Rightarrow \neg \mathbf{P_j}(\overline{\mathbf{x}})$$

resultando la función while-computable:

$$\xi(\overline{\mathbf{x}}) \cong \begin{cases} \psi_{\mathbf{1}}(\overline{\mathbf{x}}) & \mathbf{P_1}(\overline{\mathbf{x}}) \\ \ldots & \ldots \\ \psi_{\mathbf{n-1}}(\overline{\mathbf{x}}) & \mathbf{P_{n-1}}(\overline{\mathbf{x}}) \\ \psi_{\mathbf{n}}(\overline{\mathbf{x}}) & \text{c.c.} \end{cases}$$

LEY DE LA FUNCIÓN INVERSA

La INVERSA de una función while-computable y total es también while-computable

- Si la función total e **inyectiva**:

$$\mathbf{f}: \Sigma^* \longrightarrow \Sigma^*$$

es while-computable, entonces también es while-computable su función INVERSA:

$$\mathbf{f^{-1}}: \Sigma^* \longrightarrow \Sigma^*$$

- Demostración:

```
AUX := ε;
while f(AUX) /= X1  loop
      AUX := sig(AUX);
end loop;
X0 := AUX;
```