

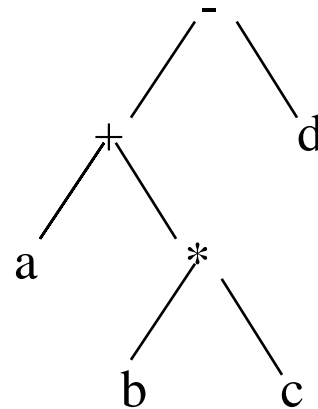
Código Intermedio

Usos del Código Intermedio

- Múltiples lenguajes y compiladores
 - $n+m$ módulos $\rightarrow n*m$ compiladores
- Optimización
 - Representación de fácil modificación
- Modelos de generación de código o Máquinas abstractas
 - FAM, G-machine, máquina de Warren
- Compiladores transportables
 - PASCAL /P-Code, JAVA
- Interpretación rápida
 - OAKLisp, CLISP
- Depuración
 - Intérpretes de C u otros lenguajes

Notación polaca Inversa (postfija)

- Se utiliza principalmente para la representación de expresiones aritméticas
- Expresión a notación polaca inversa
 - Algoritmo
 - Representar la expresión en forma de árbol sintáctico
 - Recorrer el árbol en postorden
 - Ejemplo: $a+b*c-d$



Código: $a b c * + d -$

Extensión a instrucciones de control de flujo

- Ejemplo:
 - Fuente: if <cond> then <instr1> else <instr2>
 - No se puede traducir a: <cond> <instr1> <instr2> If ya que solo se ha de evaluar una de las instrucciones
 - Código generado

```
<cond>
etiqueta1
Salta si falso
<instr1>
etiqueta2
Salta
etiqueta1:
<instr2>
etiqueta2:
```

Código más típico

```
<cond>
Salta Si falso Etiqueta1
<Instr1>
Salta Etiqueta2
Etiqueta1:
<Instr2>
Etiqueta2:
```

Pros y Contras de la Notación Polaca Inversa

- Generación de código:
 - Simple
 - No utiliza registros
- Optimización
 - Es difícil de reordenar ya que hay que considerar el contenido de la pila
- Interpretación rápida
 - Es muy fácil de interpretar ya que solo necesita una pila
- Transportable:
 - Si, ya que todos los procesadores implementan una pila.

n-tuplas

- En la notación de n-tuplas cada instrucción es una n-tupla.
 - Tripletas
 - $\langle \text{operador} \rangle, \langle \text{operando1} \rangle, \langle \text{operando2} \rangle$
 - El resultado se asocia al número de tripleta
 - Tripletas indirectas
 - Son tripletas donde el orden de ejecución se especifica a parte.
 - Cuadruplas
 - $\langle \text{operación} \rangle, \langle \text{operando1} \rangle, \langle \text{operando2} \rangle, \langle \text{resultado} \rangle$

Tripletas

- Tripleta: $\langle \text{operador} \rangle$, $\langle \text{operando1} \rangle$, $\langle \text{operando2} \rangle$
el resultado se asocia al número de tripleta
 - Ejemplo: $W * X + (Y + Z)$
 1. $*$, W , X
 2. $+$, Y , Z
 3. $+$, (1), (2)
 - Control de flujo:
 $\text{IF } X > Y \text{ THEN } Z = X \text{ ELSE } Z = Y + 1$
 1. $>$, X , Y
 2. Saltar si falso, (1), 5
 3. $=$, Z , X
 4. Saltar, , 7
 5. $+$, Y , 1
 6. $=$, Z , (5)
 7. ...
 - Problema: La optimización supone mover tripletas y hay que recalcular las referencias

Tripletas Indirectas

- Tripletas indirectas. Solucionan el problemas de la reordenación mediante indirección
- Ejemplo:

– Fuente:

$$A=B+C*D/E$$

$$F=C*D$$

– Operaciones

Tripletas

1. (1)

(1) *, C, D

2. (2)

(2) /, (1), E

3. (3)

(3) +, B (2)

4. (4)

(4) =, A, (3)

5. (1)

(5) =, F, (1)

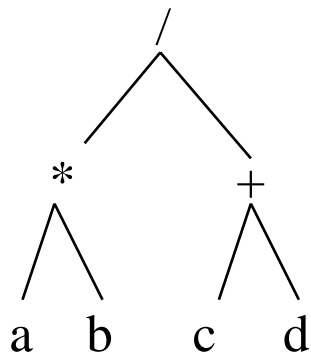
6. (5)

Cuádruplas

- Cuádrupla:
 <operación>, <operando1>, <operando2>, <resultado>
- Ejemplo:
 $(A+B)*(C+D)-E$
 +, A, B, T1
 +, C, D, T2
 *, T1, T2, T3
 -, T3, E, T4
 T1, T2, T3, T4 son variables temporales.
- Las cuádruplas facilitan la aplicación de muchas optimizaciones, pero hay que tener un algoritmo para la reutilización de las variables temporales (reutilización de registros del procesador).

Árboles sintácticos abstractos

- Fácil creación:
 - Se crea a partir del árbol sintáctico. Se elimina la información innecesaria.
 - Símbolo no terminales
 - Ejemplo: $(a*b)/(c+d)$



Árboles sintácticos abstractos

- Fácil optimización:
 - El árbol representa directamente la estructura del lenguaje fuente, y por tanto, las modificaciones que haríamos en el fuente son fáciles de realizar en el árbol.
 - Como el árbol representa la estructura del lenguaje es fácil aplicarle las propiedades de este para optimizar.
 - Propiedad conmutativa.
 - Propiedad asociativa.
 - etc.

Ejemplos de Árbol en CoSeL

- Un árbol sintáctico esta formado por las estructuras apply con el contenido:
 - Función: operador raíz
 - Arg(0), Arg(1)... Argumentos del operador
 - Ejemplo:

```
VerArbol(`(
  Fun f(n)=>
    if (n<=2) 1
    else f(n-2)+f(n-1)
  ))
```

```
Fun
+- []
+- =>
  +- f
  | +- n
+- If_Else
  +- <=
  | +- n
  | +- 2
+- 1
+- +
  +- f
  | +- -
  | +- n
  | +- 2
+- f
  +- -
  +- n
  +- 1
```

Buscar las Variables en un Árbol

```
Fun Variables(arbol)=>
{
  if (TypeP(Symbol,arbol)) [arbol]
  else if (TypeP(Apply,arbol)) {
    Var lv=[]; // lista de variables
    for (i<-arbol:Index)
      lv=lv | Variables(arbol.arg(i));
    lv
  }
  else []
}
variables(`( a=b+c; d=a*2; ))
[d,a,c,b]
```

Buscar las Variables Modificadas en un Árbol

```
Fun VariablesModificadas(arbol)=>
{
  if (ApplyFunP(arbol,`\=,2) &&
      TypeP(Symbol,arbol.arg(0))) {
    [arbol.Arg(0)]
  }
  else if (TypeP(Apply,arbol)) {
    Var lv=[];
    for (i<-arbol:Index) lv=lv |
      VariablesModificadas(arbol.arg(i));
    lv
  }
  else []
}
VariablesModificadas(`(a=b+c;d=a*2;))
[d,a]
```

Eliminar Expresiones Redundantes (NO FUNCIONA CORRECTAMENTE)

```
Type Redundante(variable,expresion);
Fun EliminarRedundantes(arbol)=> {
  Var ExpR=[];
  Fun Eliminar(a)=> {
    Search (r<-ExpR,r.expresion==a) r.variable
  else {
    if (TypeP(apply,a)) {
      Var na=newapply(a.function,a.nargs);
      for (i<-a:index)
        na.arg(i)=Eliminar(a.arg(i));
      na;
    } else a;
  }
}
...

```

Eliminar Expresiones Redundantes (NO FUNCIONA CORRECTAMENTE)

...

```
Fun Buscar(a)=> {
  if (ApplyFunP(a, `\=,2)) {
    Var ne=Eliminar(a.arg(1));
    ExpR=Redundante(a.Arg(0),ne)::ExpR;
    << <a.Arg(0)> = <ne> >>
  } else if (TypeP(Apply,a)) {
    Var na=newapply(a.function,a.nargs);
    for (i<-a:index)
      na.arg(i)=Buscar(a.arg(i));
    na;
  } else a
}
Buscar(arbol);
}
```

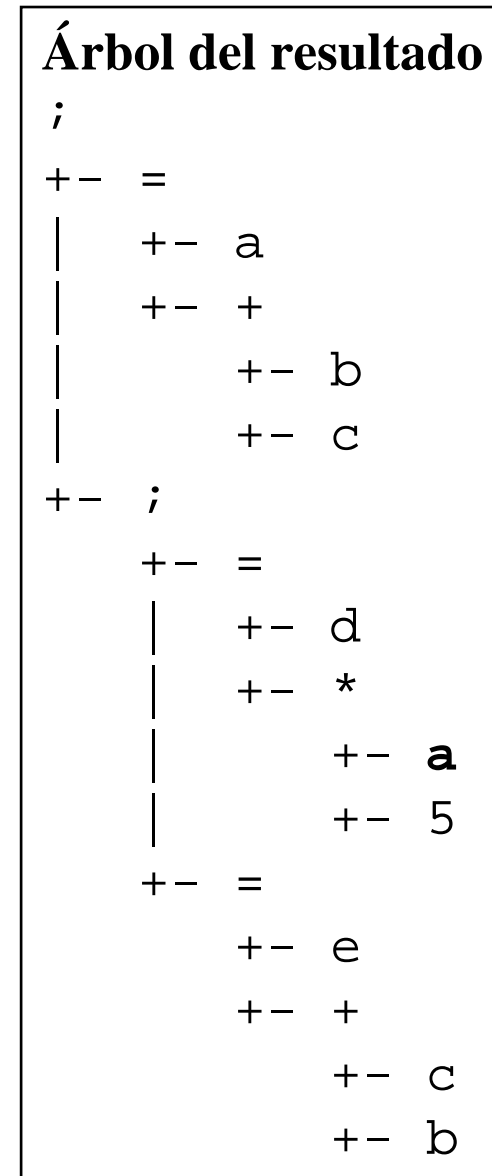
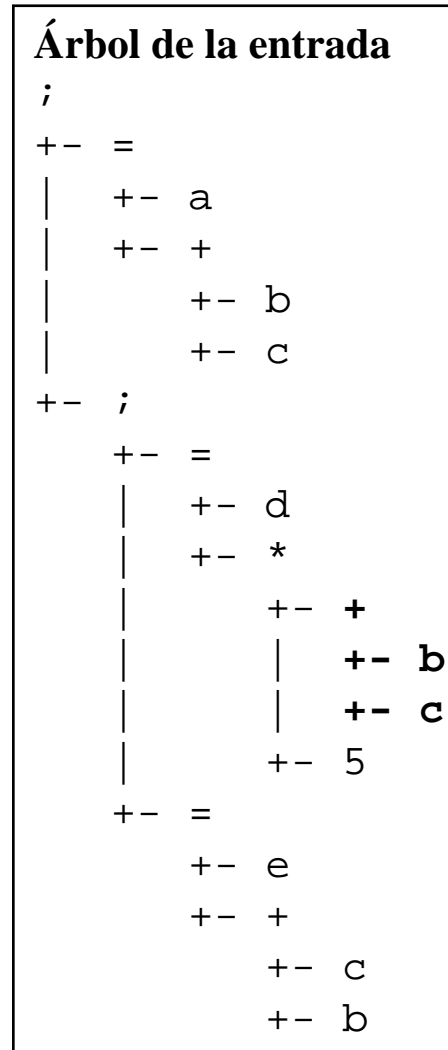
Eliminar Expresiones Redundantes

- Ejemplo**

```
EliminarRedundantes(`(
    a=b+c;
    d=(b+c)*5;
    e=c+b;
))
```

- Resultado**

```
a=b+c;
d=a*5;
e=c+b;
```



Listas en CoSeL

- Tipo de datos

```
Type Cons(Head,Tail)
```

```
Type List=Cons | []
```

- Construcciones equivalentes

```
[1,2,3]==cons(1,cons(2,cons(3,[])))
```

```
Cons(h,t)==h::t
```

```
[1,2,3]==1::2::3::[]
```

```
[h1,h2...::t]==h1::h2::...::t
```

Listas en CoSeL

- Operaciones

`[1, 2, 3].Head==1`

`[1, 2, 3].Tail==[2, 3]`

`[1, 2, 3].Head(0)==1`

`[1, 2, 3].Head(1)==2`

`[1, 2]++[2, 3]==[1, 2, 2, 3]` (append)

`[1, 2] | [2, 3]==[1, 2, 3]` (unión)

`[1, 2] & [2, 3]==[2]` (intersección)

`[1, 2] - [2, 3]==[1]` (resta)

- Listas por comprensión

`[x | x < -1..5]==[1, 2, 3, 4, 5]`

`[x | x < -[1, 2, 3, 4, 5], x%2==0]==[2, 4]`

Árboles

- Estructura
 - Nodo del arbol: `Type apply(function, arg...)`
 - Hoja: Cualquier tipo de datos que no sea `apply`
- Construcción
 - Árbol en forma de literal
 - ``(1+2+A) →`
`apply(\+, apply(\+, 1, 2), A) == 1+2+A`
 - Árbol en forma de patrón
 - `Var a=10;`
`<< 1+A+<A> >> → apply(\+, apply(\+, 1, A), 10) == 1+A+10`
 - Árbol construido directamente
 - `Var a=10;`
`apply(`\+, apply(`\+, 1, `A), A) →`
`apply(\+, apply(\+, 1, A), 10) == 1+A+10`

Estructuras de Control

- Instrucciones de control de flujo
 - Típicas
 - if** (*condición*) *instrucción*
 - if** (*condición*) *sentencia* **else** *sentencia*
 - switch** (*selector*) {*casos*}
 - while** (*condición*) *instrucción*
 - do** *instrucción* **while** (*condición*)
 - Break** [*condición*]
 - Continue** [*condición*]
 - No típicas
 - for** (*generadores*) *instrucción*
 - Search** (*generadores*) *instrucción*
 - Search** (*generadores*) *sentencia* **else** *sentencia*
 - Found** (*generadores*) *sentencia*

Estructuras de Control

- Generadores

```
variable <- lista
```

```
variable <- vector : Index
```

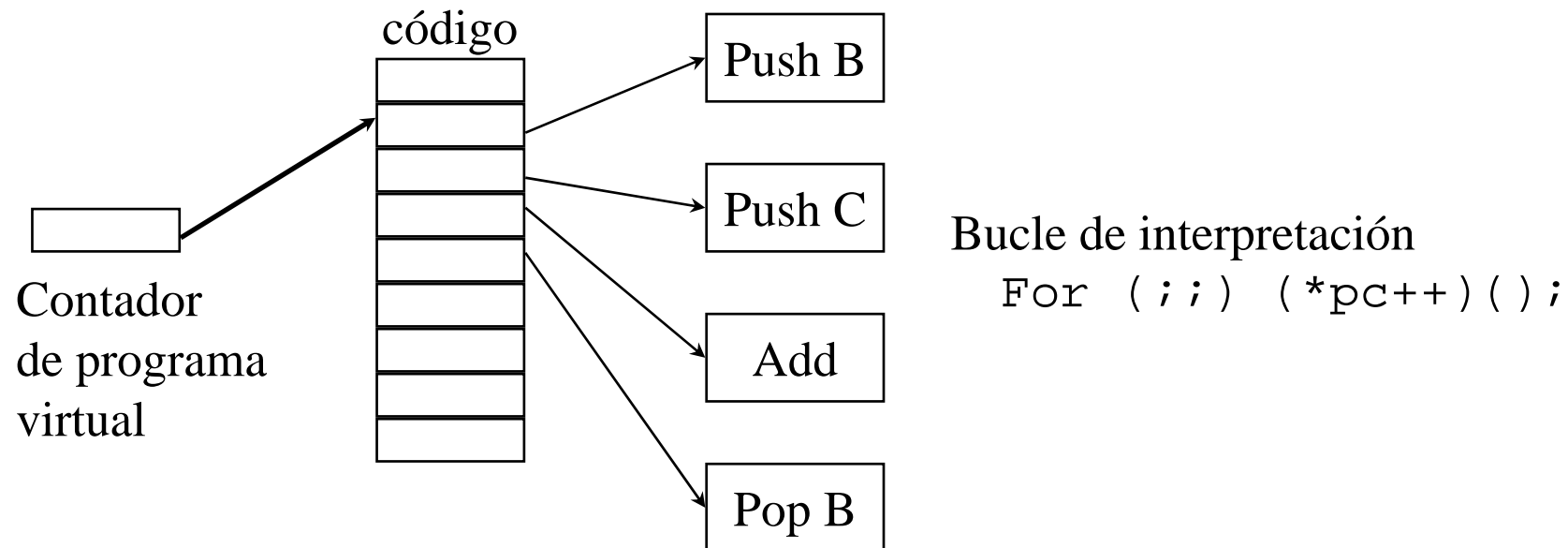
```
variable <- apply : Index
```

```
variable <- mínimo .. máximo
```

```
variable <-~ mínimo .. máximo
```

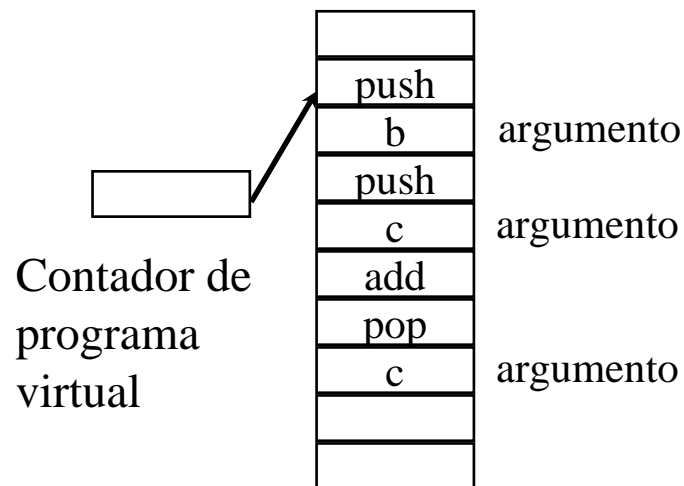
Código Enhebrado

- Es una representación útil para la interpretación independiente de máquina.
- Código enhebrado directo: El código es una secuencia de direcciones de rutinas que implementan las instrucciones.
 - Algunas rutinas son estándar (Add), pero otras las ha de generar el compilador para que puedan tratar los operandos (Push). Estas últimas limitan la transportabilidad.



Código Enhebrado con Parámetros

- Código enhebrado indirecto: El código es una secuencia de apuntadores a funciones seguidos de los argumentos que pueda necesitar la función.



Bucle de interpretación

```
For ( ; ; ) ( *PC ) ( ) ;
```

Función

```
void pushLiteral()  
{  
    *--SP=PC[1];  
    PC=PC+2;  
}
```

Código para máquinas abstractas

- Un objetivo de los compiladores es ser adaptables y portables a nuevas máquinas. Una de las formas de conseguirlo es con la generación de código para máquinas abstractas.
- Aplicaciones:
 - Modelos de compilación
 - WAM modelo de compilación de PROLOG
 - G-machine modelo de compilación de lenguajes funcionales con evaluación perezosa.
 - FAM modelo de compilación de lenguajes funcionales con evaluación estricta.

Código para máquinas abstractas

- Compiladores portables
 - Compilador ha código intermedio + intérprete.
 - Compilador ha código intermedio + compilador de código intermedio a código máquina
 - $n+m$ módulos $\rightarrow n*m$ compiladores

Resumen

- Códigos para la ejecución
 - Notación polaca inversa
 - Código enhebrado
- Códigos para la optimización
 - Árbol sintáctico abstracto
 - Tuplas
 - Tripletas
 - Tripletas indirectas
 - Cuadruplas
- Códigos para la explicación
 - Códigos de máquinas abstractas