UNIVERSIDAD
FRANCISCO DE VITORIA
VINCE IN BONO MALUM

PROCESADORES DE LENGUAJE

Tema 4: Implementación de un lenguaje imperativo simplificado

Departamento de Ingeniería Informática
Universidad Francisco de Vitoria

Curso 2009/10

Objetivos del tema

UNIVERSIDAD FRANCISCO DE VITORIA

- Objetivos
  1. Conocer ventajas e inconvenientes del uso de representaciones intermedias
  2. Explicar el funcionamiento de la tabla de símbolos y su modificación durante las etapas de análisis léxico y sintáctico.
  3. Conocer los tipos de representaciones intermedias que se manejan:
     1. Notación postfija
     2. Máquinas de pila
     3. Código de 2 direcciones
     4. Código de 3 direcciones. Implantación en tercetos y/o cuartetos
  4. Saber entender el código intermedio como un atributo sintetizado del análisis semántico
  5. Saber generar código intermedio (para una máquina pila).

Contenido

UNIVERSIDAD FRANCISCO DE VITORIA

- Compiladores de un solo paso: Fases de análisis.
- Tabla de Símbolos y restricciones contextuales.
- Lenguajes intermedios y máquinas abstractas.
- Generación de código para una máquina pila.

Bibliografía recomendada

UNIVERSIDAD FRANCISCO DE VITORIA

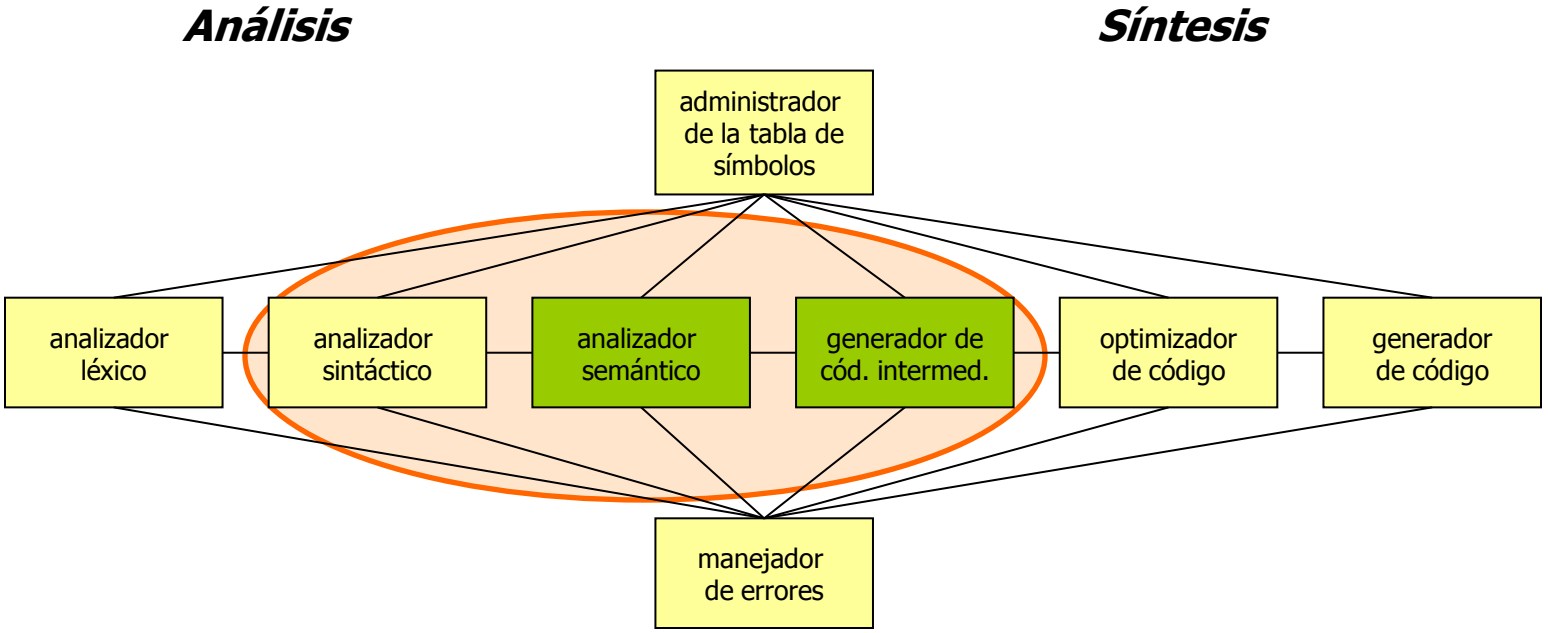
- ***Compiladores*** (Principios, técnicas y herramientas)
  PEARSON, Addison Wesley Longman
  - Capítulo 8: Generación de código intermedio

1. Compiladores de un solo paso: Fases de análisis.

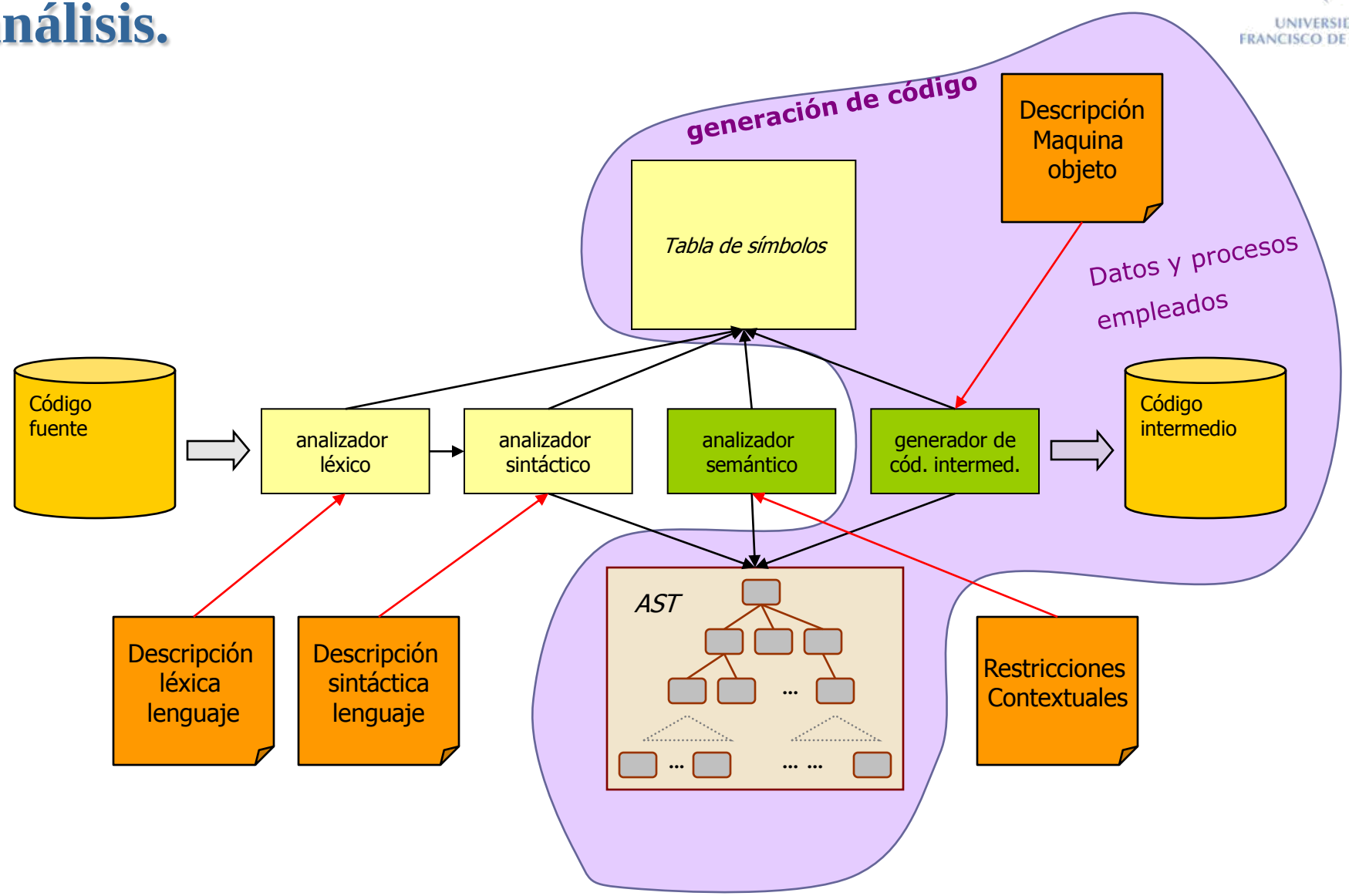
UNIVERSIDAD FRANCISCO DE VITORIA

Introducción

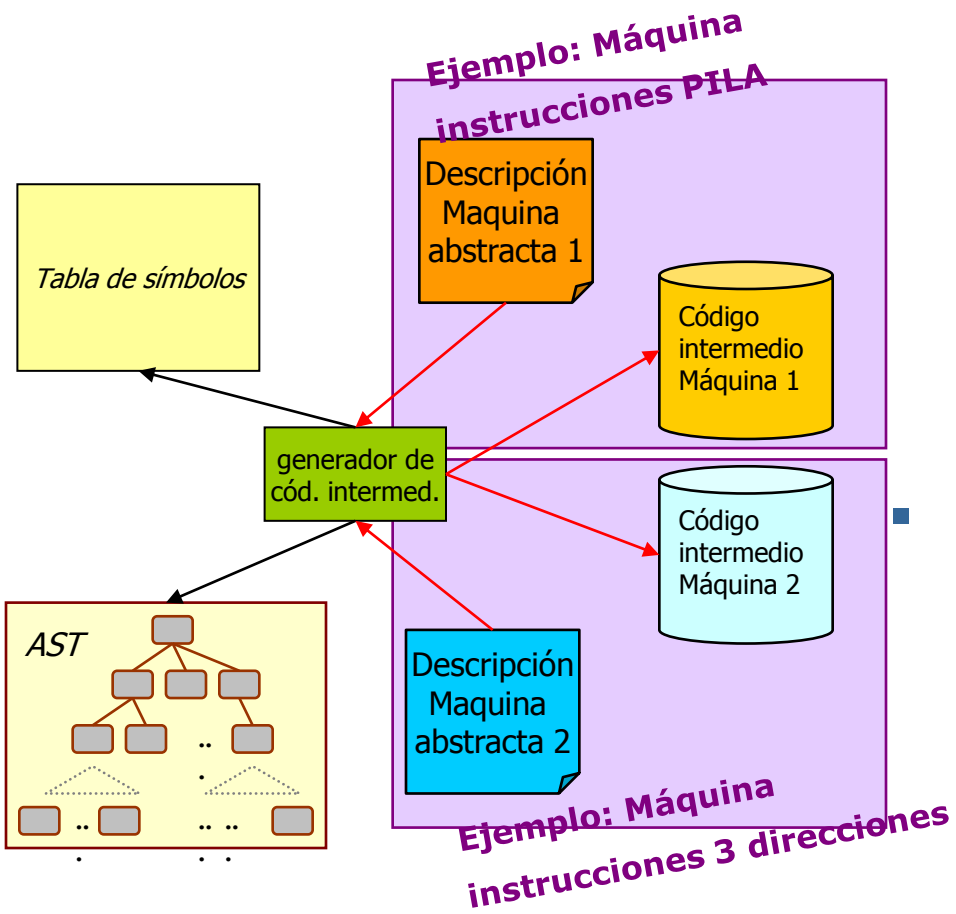
- Los compiladores de una pasada permiten la generación de código al mismo tiempo que se realiza la construcción del árbol sintáctico y se comprueban las restricciones contextuales.



1. Compiladores de un solo paso: Fases de análisis.



1. Compiladores de un solo paso: Fases de análisis.



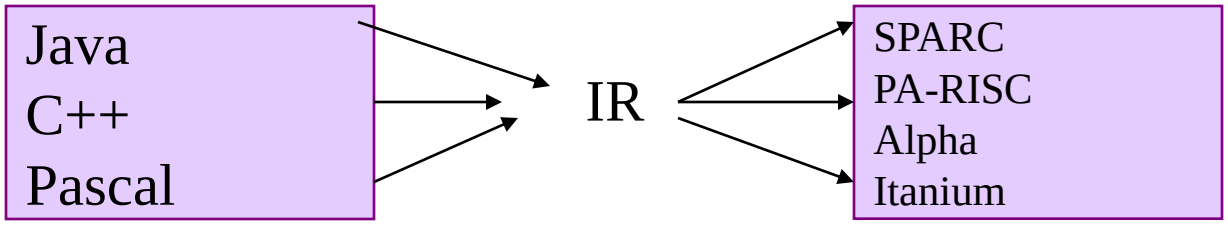
- Una misma representación intermedia de árbol sintáctico y tabla de símbolos se puede traducir a distintos códigos objeto intermedios en función de las características del tipo de máquina abstracta que vaya a evaluar las instrucciones.
- La mayoría de los compiladores transforman el programa fuente en algún tipo de representación intermedia (versión del código fuente independiente del lenguaje de la máquina objeto), para luego convertir esa representación en otra más eficiente o en el código final.

1. Compiladores de un solo paso: Fases de análisis.

UNIVERSIDAD FRANCISCO DE VITORIA

Representaciones intermedias

- Aunque esas representaciones intermedias son 'un paso más', ofrecen ventajas que aconsejan su uso:
  1. Independizar el front-end (dependiente del lenguaje) del back-end (dependiente de la máquina), con lo que eso supone en cuanto a modularidad y a posibilidades para compilaciones cruzadas (portabilidad).
  2. Mayor nivel de abstracción.
  3. Posibilidad de realizar optimizaciones: la mayor parte de las optimizaciones se realizan sobre representaciones intermedias del código.
- Las representaciones intermedias (su tipo, diseño…) afectan tanto la velocidad como la eficiencia del compilador.



1. Compiladores de un solo paso: Fases de análisis.

UNIVERSIDAD FRANCISCO DE VITORIA

Representaciones intermedias: Ventajas

1. Mayor modularidad. (Ofrece abstracción para componentes de alto nivel)
   - Aísla elementos de más alto nivel de los dependientes de la máquina
2. Facilita optimización y generación de código
   - Elimina/simplifica características específicas de la máquina objetivo:
     - Nº limitado de registros, tipos de instrucciones, alineación de datos, modos de direccionamiento, etc,...
   - Permite optimizaciones independientes de la máquina
   - Ofrece representación simple y uniforme fácil generar/ optimizar código
3. Mayor portabilidad
   - Independencia de la máquina objetivo y del lenguaje fuente
   - Análisis no depende de arquitectura destino
   - Generación código no depende del lenguaje original
   - Crear un compilador para otra máquina basta crear una nueva etapa final de generación
   - Ejemplo: multicompiladores

1. Compiladores de un solo paso: Fases de análisis.

UNIVERSIDAD FRANCISCO DE VITORIA

Representaciones intermedias: Inconvenientes

1. Necesidad de una fase extra para traducir a código máquina
   1. Mayor coste computacional
   2. Dificultad para definir un lenguaje intermedio adecuado

2. Compromiso entre la representación de elementos código fuente y del código máquina.

1. Tabla de símbolos. Restricciones contextuales

UNIVERSIDAD FRANCISCO DE VITORIA

- La tabla de símbolos es una estructura fundamental durante todas las etapas del compilador.
- Sirve como base de datos temporal y almacena todos los atributos asociados a los lexemas (tokens) del código de entrada, los cuales harán falta durante la compilación.
- La incorporación de atributos a la tabla de símbolos es realizada por las distintas fases del compilador (léxica, sintáctica y semántica).
- Los atributos asociados a clases de tokens diferentes pueden ser distintos puesto que el propósito y semántica de ellos también lo son:
  - Ejemplo: Un token ID correspondiente a una variable, requerirá, entre otros atributos, alguno que indique la posición y rango de memoria a ocupar dentro del código objeto. Un token de palabra reservada no requerirá dicho atributo de memoria.

1. Tabla de símbolos. Restricciones contextuales

hora	Incorporado por análisis **léxico**
VARIABLE	Incorporado por análisis **semántico**

```
...
var hora : int ;
...
```

En el analizador léxico puede incorporar estos tres atributos para cualquier token:
- Tipo de token
- Identificador numérico interno secuencial
- Lexema del token

TOKEN_TYPE	*TOKEN_NUM*	*TOKEN_LEX*	*ID_CLASS*	*ID_TYPE*	*MEMDIR*	*MEMSIZE*	*...*
...							
PAL_VAR	23	var	NA	NA	NA	NA	
ID	**24**	**hora**	**VARIABLE**	**ENTERO**	**10**	**1**	
SIMB_2PTOS	25	:	NA	NA	NA	NA	
PAL_INT	26	int	NA	NA	NA	NA	
...							

No todos los atributos son aplicables a todos los tipos de tokens

La etapa sintáctica y semántica pueden agregar más información en función de su posición en el árbol AST o sus restriciones contextuales (relación entre atributos de la gramática)

1. Tabla de símbolos. Restricciones contextuales

UNIVERSIDAD FRANCISCO DE VITORIA

Ejemplo:

D → var L : T ;

L → ID , L

L → ID

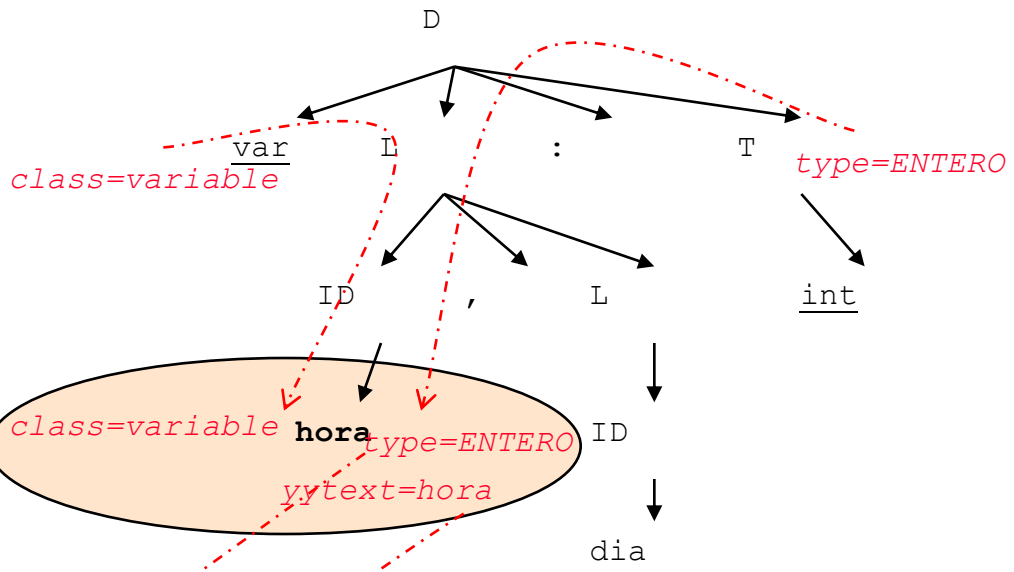
T → int

T → real

F → function ID (A) : T

...

w = var hora, dia : int ;



AñadeAtribEnTS(TS, token_num, "ID_CLASS", ID.class);

TOKEN_TYPE	TOKEN_NUM	TOKEN_LEX	ID_CLASS	ID_TYPE	MEMDIR	MEMSIZE
ID	24	hora	variable	ENTERO		

CreaEnTS(TS, "ID");
+
AñadeAtribEnTS(TS,token_num,"TOKEN_LEX",yytext)

1. Tabla de símbolos. Restricciones contextuales

Ejemplo interacción T.S. durante **reconocimiento léxico**:

```
"var"                 {token_num=CreaEnTS(TS, "PAL_VAR");
                       AñadeAtribEnTS(TS,token_num,"TOKEN_LEX","var");
                       return PAL_VAR};
"int"                 {token_num=CreaEnTS(TS, "PAL_INT");
                       AñadeAtribEnTS(TS,token_num,"TOKEN_LEX","int");
                       return PAL_INT};
"real"                {token_num=CreaEnTS(TS, "PAL_REAL");
                       AñadeAtribEnTS(TS,token_num,"TOKEN_LEX","real");
                       return PAL_REAL};
[a-zA-Z][a-zA-Z0-9]*  {token_num=CreaEnTS(TS, "ID");
                       AñadeAtribEnTS(TS,token_num,"TOKEN_LEX",yytext);
                       return ID};
":"                   {token_num=CreaEnTS(TS, "SIMB_2PTOS");
                       AñadeAtribEnTS(TS,token_num,"TOKEN_LEX",":");
                       return SIMB_2PTOS};
```

La función "CreaEnTS" añade una nueva entrada para el token reconocido y asigna valores a las columnas TOKEN_TYPE y TOKEN_NUM. Devuelve el valor de TOKEN_NUM como referencia interna (descriptor) de dicho elemento.

"AñadeAtribEnTS" agrega, en este caso, el atributo TOKEN_LEX a la entrada generada en el paso anterior. Asigna la cadena de caracteres con la que se corresponde.

1. Tabla de símbolos. Restricciones contextuales

Ejemplo interacción T.S. durante **reconocimiento sintáctico**:

```
D → var L : T ;  { L.class="VARIABLE"; L.type=T.type }
L → ID , L       { ID.class=L.class; ID.type=L.type;
                   L2.class=L.class; L2.type=L.type;
                   token_num=BuscaEnTS(TS,ID);
                   AñadeAtribEnTS(TS, token_num, "ID_CLASS", ID.class);
                   AñadeAtribEnTS(TS, token_num, "MENDIR", ReservaEspacio(ID.class));
                 }
L → ID           { ID.class=L.class;
                   token_num=BuscaEnTS(TS,ID);
                   AñadeAtribEnTS(TS, token_num, "ID_CLASS", ID.class);
                   AñadeAtribEnTS(TS, token_num, "MENDIR", ReservaEspacio(ID.class));
                 }
T → int          { T.type="ENTERO" }
T → real         { T.type="REAL" }

F → function ID ( A ) : T
                 { ID.class="FUNCTION"; ID.type=T.type;
                   F.class="FUNCTION"; F.type=T.type;
                   token_num=BuscaEnTS(TS,ID);
                   AñadeAtribEnTS(TS, token_num, "ID_CLASS", ID.class);
                   AñadeAtribEnTS(TS, token_num, "MENDIR", ReservaEspacio(ID.class));
                   ...
```

La gramática de atributos manejada durante el reconocimiento sintáctico/semántico permite calcular el valor de nuevos atributos asociados a los token y necesarios para reconocer generar el código. Tales como: clase de token (un ID puede ser variable, función, objeto, etc...), tipo asociado, ubicación y tamaño en memoria (en función de la clase y tipo), etc ...

"AñadeAtribEnTS" agrega, en estos casos, los atributos ID_CLASS y MEMDIR, que pueden ser calculados en este momento.

1. Tabla de símbolos. Restricciones contextuales

UNIVERSIDAD FRANCISCO DE VITORIA

- Generalmente, las Tablas de símbolos se manejan como estructuras globales disponibles en cualquier fase del compilador.
- Se puede optar por el uso de varias tablas de símbolos independientes cuando es necesario vincular ciertos tokens solo a algunas partes del código (por ejemplo, tokens de variables locales en procedimientos y funciones).

Ejemplo:

```
var a : int;
function suma ( op1 : int , op2 : int) : int
    var a : int;
    begin
        a:= op1 + op2;
        return a;
    end
begin
    a:=suma(3,4);
end
```

Estas dos variables "a" son independientes. Durante el diseño del compilador, debe decidirse si se pueden mantener en una misma (única tabla de símbolos) o si se deben definir tablas de símbolos independientes para procedimientos, funciones, clases, etc...

1. Tabla de símbolos. Restricciones contextuales

UNIVERSIDAD FRANCISCO DE VITORIA

TS global

TOKEN_TYPE	*TOKEN_NUM*	*TOKEN_LEX*	*ID_CLASS*	*ID_TYPE*	...	*PTO_TS_FUNC*
ID	23	a	variable	ENTERO		
ID	24	suma	function	ENTERO	...	

TS local function suma

TOKEN_TYPE	*TOKEN_NUM*	*TOKEN_LEX*	*ID_CLASS*	*ID_TYPE*	...	*PTO_TS_FUNC*
ID	102	a	variable	ENTERO		

3. Lenguajes intermedios y máquinas abstractas.

UNIVERSIDAD FRANCISCO DE VITORIA

Representaciones intermedias: tipos

- Hay varios tipos de representaciones intermedias y varias formas de clasificarlas.
- Se diferencian en lo más o menos cercanas que estén del código máquina o del lenguaje fuente: hay RI de alto nivel de nivel medio y de bajo nivel, que 'contienen' distinto tipo de información, y cada tipo será adecuado para algunas optimizaciones.
  - Las **RI de alto nivel** preservan información como la de los bucles y las sentencias if-then-else. Reflejan el lenguaje fuente que están compilando.
  - Las de **nivel medio** son independientes tanto de la estructura del lenguaje. fuente como del micro para el que se están compilando.
  - Las de **bajo nivel** se acercan a la arquitectura 'objetivo': dependen mucho de la máquina, aunque no llegan a ser 'código ensamblador'.
- Es normal que un compilador comience generando una RI de alto nivel, realice algunas optimizaciones, traduzca esa RI en otra de nivel medio y vuelva a optimizar y finalmente genere otra RI de bajo nivel para una última ronda de optimizaciones. Todo ello antes de generar el código final.

3. Lenguajes intermedios y máquinas abstractas.

UNIVERSIDAD FRANCISCO DE VITORIA

- Una representación intermedia es una estructura de datos que representa al programa fuente durante el proceso de su traducción a código objeto. Este tipo de código abstracto y virtual debería reunir las características siguientes:
  1. Ser fácil de generar a partir del árbol sintáctico (durante la fase de A. Semántico).
  2. Ser independiente del entorno de ejecución: facilitar traducción al lenguaje. máquina final para todas las posibles máquinas objetivo.
  3. Construcciones claras, sencillas y uniformes, con significado unívoco: Facilita la especificación de la traducción a cada máquina objetivo
- Tres categorías fundamentales:
  - Representaciones arbóreas o estructurales (muy usadas en compiladores cruzados. Ejem: AST, Grafos Dirigidos Acíclicos, grafos de dependencias de datos o de flujo).
  - Representaciones lineales (una forma de pseudo código para una máquina abstracta: pueden aportar diferentes niveles de abstracción, usan estructuras de datos simples y compactas… Ej: notación postfija, código 3 direcciones…)
  - Híbridas (combinaciones de grafos y pseudocódigo)

3. Lenguajes intermedios y máquinas abstractas.

UNIVERSIDAD FRANCISCO DE VITORIA

Ejemplos de RI con 'nombre propio"

- Diana: representación arbórea usada en compiladores de ADA.
- RTL (register transfer language): usada en familia de compiladores GCC (GNU Compiler Collection o Colección de compiladores GNU)
- Código-P: compiladores Pascal
- WAM (Warren abstract machinne): intérpretes Prolog.
- Bytecodes JAVA.

UNIVERSIDAD FRANCISCO DE VITORIA

Elementos de RIs de Medio Nivel

- Proporcionan las siguientes abstracciones:
  - Variables del código fuente
  - Variables temporales
  - El flujo de ejecución se realiza con saltos, llamadas a funciones y retornos de éstas
  - Asignación de variables
  - Obtención de direcciones (&) e indireccionamiento (*)
  - Puede utilizarse una pila de ejecución y/o un conjunto de registros simbólicos
- Las representaciones más empleadas son:
  - Máquinas de pila
  - Código de dos y de tres direcciones
  - Notación postfija o "polaca inversa"

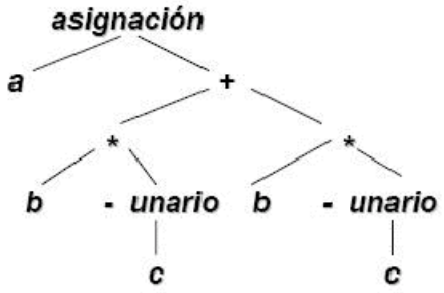
3. Lenguajes intermedios y máquinas abstractas.

UNIVERSIDAD FRANCISCO DE VITORIA

RI Lineales: Notación Postfija

- Es una representación lineal de un AST.: es una lista de los nodos del árbol en que cada nodo aparece inmediatamente después de sus hijos

  Ej: a : = b * (-c)+ b * (-c)



árbol sintáctico

a b c -unario * b c -unario * + asignación

notación postfija

- También se llama notación sufija y polaca inversa. Los operadores aparecen justo detrás de los operandos.
- Muy usado como código intermedio en intérpretes. Interpretación muy sencilla (basta una pila) Ejemplo: Postscript

UNIVERSIDAD FRANCISCO DE VITORIA

Ventajas de la notación Postfija

- Muy sencilla para expresiones aritméticas.
  - No necesita paréntesis.
  - Precedencia está implícita → valores temporales implícitos

- Interpretación/Generación de código muy simple
  - Sólo necesita una pila. Algoritmo interpretación:
    1. Recorrer lista, apilando operandos hasta llegar a un operador.
    2. Tomar los operandos necesarios de la pila y aplicarles el operador.
    3. Apilar resultado y continuar.
  - Muy usado en primeras calculadoras comerciales.

UNIVERSIDAD FRANCISCO DE VITORIA

Máquinas de Pila

- Los operadores trabajan con (operandos) y sobre (resultados) en una estructura de pila:
  - No es común el empleo de registros simbólicos
- Ventajas
  - Generar este tipo de código es sencillo
  - Su interpretación (ejecución) resulta directa empleando una estructura de pila.
- Inconvenientes
  - Las optimizaciones de código son algo más complejas, puesto que los parámetros están implícitos en la pila
  - Aunque existen microprocesadores basados en máquinas de pila (HP3000 y BurroughsB5500), en la actualidad la mayoría están basadas en registros.

UNIVERSIDAD FRANCISCO DE VITORIA

RI Lineales: Código de dos direcciones

- Las instrucciones utilizan dos direcciones de memoria o registros, de este tipo:

  x= x operador y

- Ejemplo: z=x-2*y se convierte en

$$
\begin{aligned}
t_1 &\leftarrow 2 \\
t_2 &\leftarrow \text{load } y \\
t_2 &\leftarrow t_2 * t_1 \\
z &\leftarrow \text{load } x \\
z &\leftarrow z - t_2
\end{aligned}
$$

- Se crean variables temporales para los 'nodos intermedios'
  - Ventajas: aspecto 'muy compacto'
  - Inconveniente: es una RI no adecuada para los micros 'modernos'. Lo era para el PDP-11.

UNIVERSIDAD FRANCISCO DE VITORIA

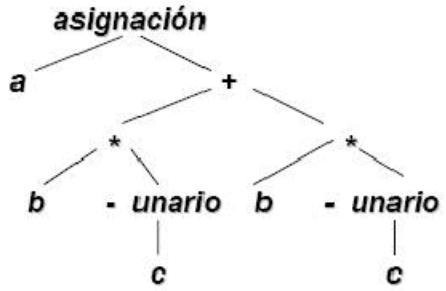
RI Lineales: Código de tres direcciones

- Las instrucciones utilizan tres direcciones de memoria o registros, dos para los operandos y una para el resultado: Esta RI es una generalización del código ensamblador de una máquina virtual de 3 direcciones.

  resultado = operando1 operador operando2

- Es la representación intermedia es la más extendida.

  Ej. a : = b * (-c)+ b * (-c) :



árbol sintáctico

```
t1 : = - c
t2 : = b * t1
t3 : = - c
t4 : = b * t3
t5 : = t2 + t4
a  : = t5
```

código de 3 direcciones

- Ventajas: adecuado para la mayoría de las máquinas, aspecto 'compacto'

3. Lenguajes intermedios y máquinas abstractas.

UNIVERSIDAD FRANCISCO DE VITORIA

RI Lineales: Código de tres direcciones

- FORMATO GENERAL : x := y OP z
- x, y, z son referencias a:
  - Nombres (dir. de variables)
  - Constantes
  - Variables temporales (creadas por el compilador durante la gen. de la R. I.)
  - Etiquetas (direcciones de instrucciones)
- Existe una referencia explícita a los resultados intermedios de las operaciones mediante las vars. temporales.
- Posibilidades de representación: cuartetos, tercetos y tercetos indirectos.

3. Lenguajes intermedios y máquinas abstractas.

UNIVERSIDAD FRANCISCO DE VITORIA

Ejemplos:

(a + b) / (c + e)	z: = x + y + z / y
t1 := a + b t2 := c + e t3 := t1 / t2	t1 := x + y t2 := t1 + z t3 := t2 / y z := t3

3. Lenguajes intermedios y máquinas abstractas.

UNIVERSIDAD FRANCISCO DE VITORIA

Tipos de instrucciones en código de tres direcciones

Instrucción	Explicación	Ejemplo
Operaciones binarias	x = y op z, en la que op es un operador binario (aritmético, lógico o relacional).	a = b and c
Operaciones unarias	x = op y, en la que op es un operador unario (negación lógica, menos unario, operadores de desplazamiento o conversión de tipos, etc.).	a = – c
Asignaciones	x = y, operación de copia en la que el valor de y se asigna a x.	a = c
Definir punto Ir al punto	label etiq, define una etiqueta. goto etiq, salto incondicional.	label T2 goto T2
Saltos condicionales	Saltos condicionales como if false x goto etiq.	if x = 0 goto T2
Ir y volver de un bloque	call f, para llamar funciones o acciones, y return, para retornar valores.	call etiqueta
Matrices (arrays) y tablas	Asignaciones con índices de la forma x = y[i], en la que se asigna a x el valor de la posición de memoria situado i unidades más allá de la posición y. También x[i] = y.	x[3] = 5
Registros	Registros Tratamiento de los campos con registro.camp, en los que el campo tiene el valor guardado a una determinada distancia del comienzo del registro.	x = y.dia
Indirecciones	Asignaciones de direcciones y apuntadores de la forma x = &y, x = *y o *x = y.	*x = 6

3. Lenguajes intermedios y máquinas abstractas.

UNIVERSIDAD FRANCISCO DE VITORIA

Ejemplo (II):

```
read x;
if 0 < x then
   fact := 1;
   repeat fact := fact * x;
      x := x - 1;
   until x = 0;
   write fact;
end;
```

```
01 read x
02 t1 = 0 < x
03 if false t1 goto L1
04 fact=1
05 label L2
06 t2 = fact * x
07 fact = t2
08 t3 = x -1
09 x = t3
10 t4 = x == 0
11 if false t4 goto L2
12 write fact
13 label L1
14 halt
```

UNIVERSIDAD FRANCISCO DE VITORIA

Ejemplo de Máquina Abstracta Pila

- A continuación de describirá un ejemplo de máquina abstracta tipo pila que servirá de referencia posteriormente durante la generación de código intermedio.
- La máquina abstracta tiene memorias independientes para las instrucciones y los datos, y todas las instrucciones aritméticas se realizan con los valores de una pila de registros.
- Las instrucciones son bastante limitadas y están comprendidas en tres clases:
  - aritmética entera: operaciones in inserción de valores fijos y operaciones aritméticas (suma, multiplicación, etc...)
  - manipulación de la pila: instrucciones para acceder a la memoria de los datos.
  - control de flujo: instrucciones que permiten alterar el orden secuencial de evaluación de instrucciones del código objeto.

3. Lenguajes intermedios y máquinas abstractas.

UNIVERSIDAD FRANCISCO DE VITORIA

INSTRUCCIONES ARITMÉTICAS

suma (+)	suma 2 operandos
resta (-)	resta 2 operandos
mult (*)	multiplicación 2 operandos
div (/)	división 2 operandos
mod (%)	resto 2 operandos
exp (^)	potencia 2 operandos
comp_and	Comparación lógica and 2 operandos
comp_or	Comparación lógica or 2 operandos

INSTRUCCIONES CONTROL DE FLUJO

etiqueta *l*	destino de salto a *l*
vea *l*	la sig. instruc. será "etiqueta *l*"
sifalsovea *l*	saca el valor tope de la pila. Salta si es 0
siciertovea *l*	saca el valor tope de la pila. Salta si es != 0
alto	detiene la ejecución

INSTRUCCIONES MANIPULACIÓN PILA

inserta *v*	inserta *v* en la pila
valord *l*	inserta el contenido de la posición de datos *l*
valori *l*	inserta la dirección de la posición de datos *l*
saca	elimina el valor de la cima de la pila
asigna (:=)	Asume que la cima contiene el valor del lado dcho y el inferior a la cima contiene la posición de memoria del lado izdo. Copia el valor de la cima la posición de memoria especificada debajo.
copia	inserta una copia del valor de la cima en la pila

3. Lenguajes intermedios y máquinas abstractas.

UNIVERSIDAD FRANCISCO DE VITORIA

INSTRUCCIONES

1	inserta 5
2	valord 2
3	+
4	valord 3
5	*
6	...
7	...
8	...

DATOS

1	0
2	4
3	8
4	2
5	7
6	...
7	...
8	...

PILA

1	16
2	8
3	8
4	2
5	7
6	
7	
8	

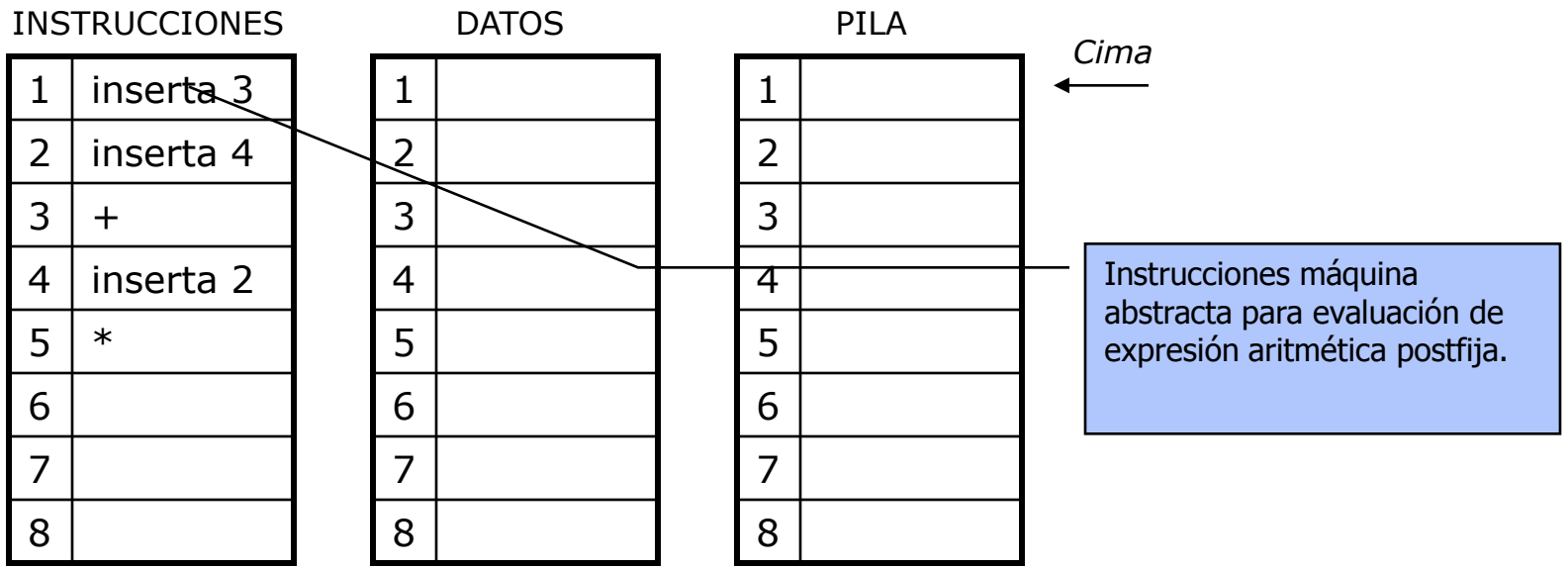
Cima ←

- El código de la máquina abstracta de una expresión aritmética **simula la evaluación postfija** de esa expresión utilizando una pila. La evaluación se realiza procesando la representación postfija de izquierda a derecha.

3. Lenguajes intermedios y máquinas abstractas.

Ejemplo:

$w=(3+4)*2 \quad \rightarrow w_{postfija} = (3\ 4\ +\ )\ 2\ *\ ==\ 3\ 4\ +\ 2\ *$

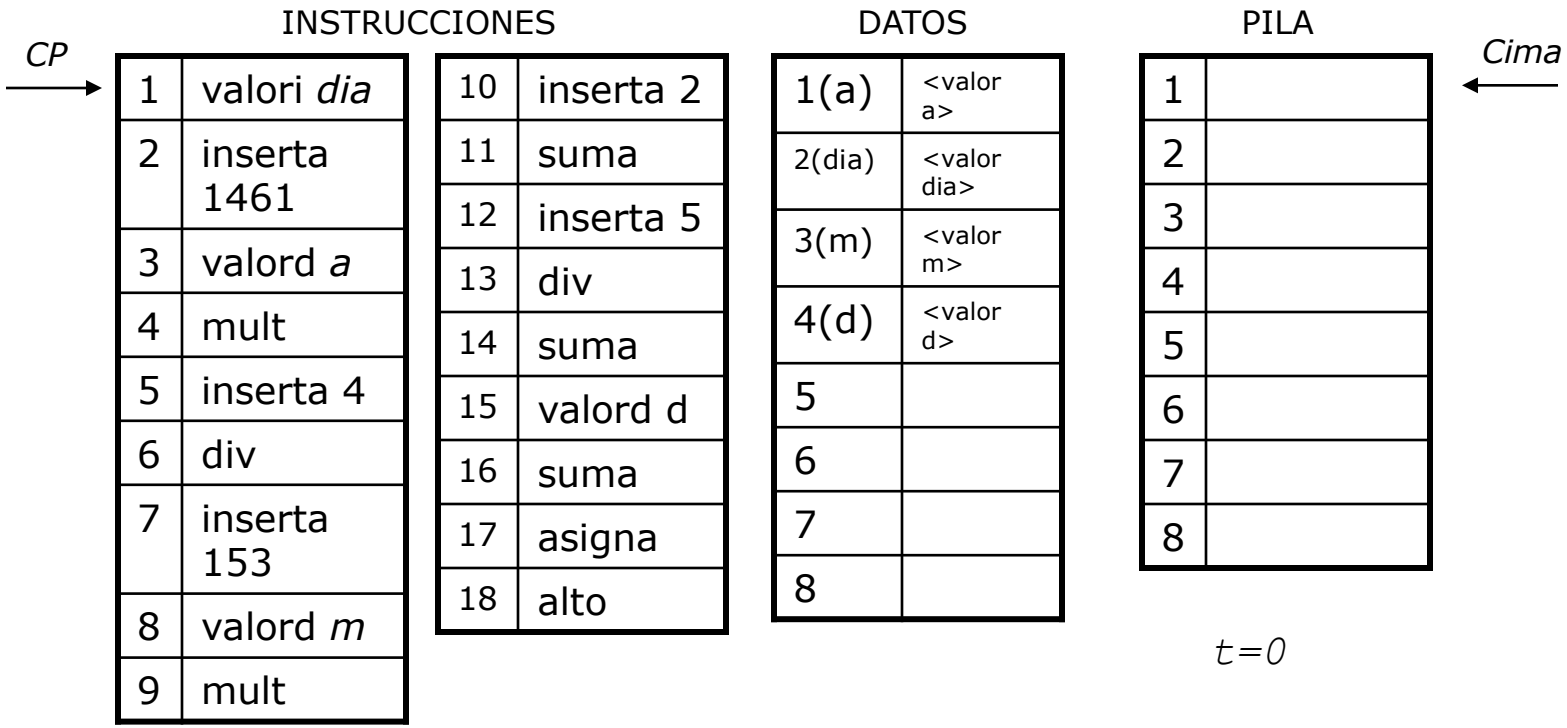


3. Lenguajes intermedios y máquinas abstractas.

Ejemplo:

*dia=(1461*a)/4 + (153*m +2)/5 + d*

*rep. postfija → dia 1461 a * 4 / 153 m * 2 + 5 / + d + =*



3. Lenguajes intermedios y máquinas abstractas.

UNIVERSIDAD FRANCISCO DE VITORIA

- Traducción de proposiciones (condicionales): Las siguientes figuras muestran el código de una máquina abstracta para las proposiciones condicionales if y while.

```
if expr then {
   prop1
}
```

código de *expr*
sifalsovea *etiq*
código *prop1*
etiqueta *etiq*

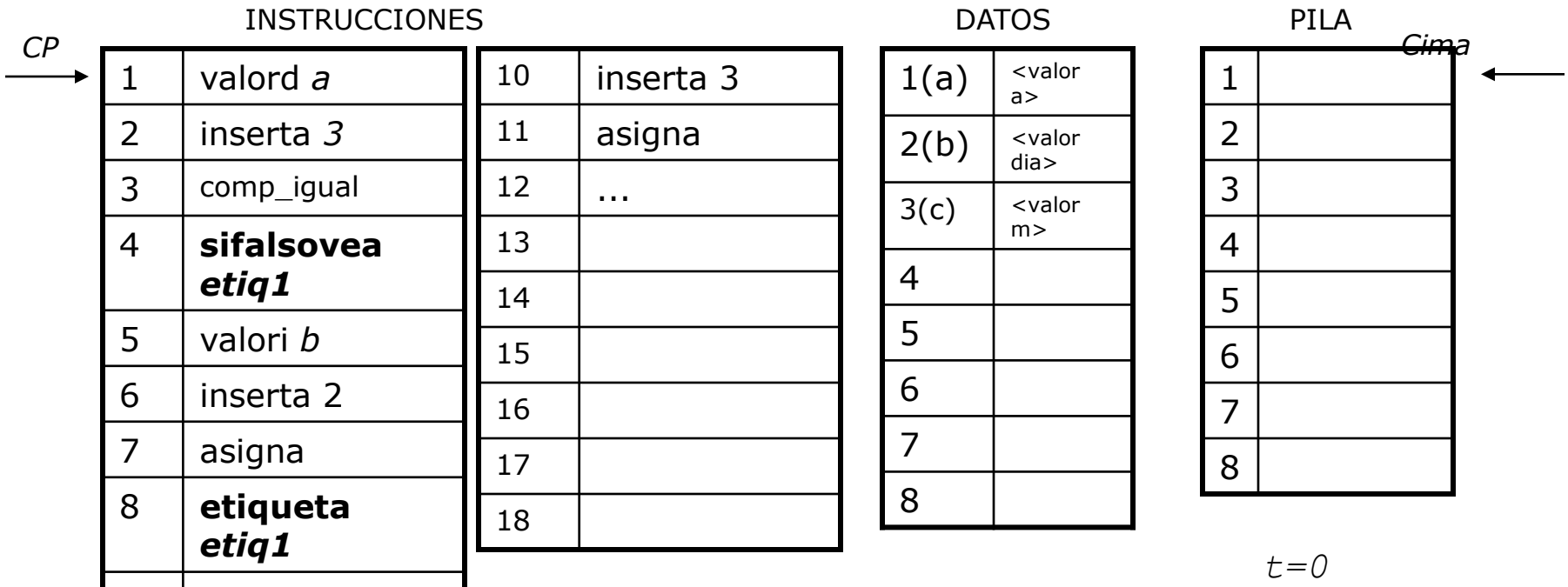
```
while expr {
   prop1
}
```

etiqueta *inicio*
código de *expr*
sifalsovea *fin*
código *prop1*
vea *inicio*
etiqueta *fin*

3. Lenguajes intermedios y máquinas abstractas.

Ejemplo:

```
if (a == 3) then {
   b:=2
}
c:=3; ...
```



3. Generación de código para una máquina pila.

UNIVERSIDAD FRANCISCO DE VITORIA

Código intermedio como atributo sintetizado

- El código intermedio puede ser considerado como un atributo sintetizado.
- El código intermedio es visto como una cadena de caracteres y se puede diseñar un esquema de traducción dirigido por la sintaxis (ETDS) que genere dicho código al recorrer el árbol de análisis sintáctico.
- Cada no terminal tendrá un atributo E.cod, que contendrá la serie de todas las sentencias de código de pila que calculan E.

3. Generación de código para una máquina pila.

UNIVERSIDAD FRANCISCO DE VITORIA

Instrucción de asignación.

DDS (Definición dirigida por sintaxis)

```
L → L ; Ia      { L.cod = L2.cod || Ia.cod }
    | e         { L.cod = "" }

Ia → ID := E    { token_num = BuscaTokenEnTS(TS,ID);
                  memdir = BuscaAtribEnTS(TS,token_num,"MEMDIR");
                  Ia.cod = "valori memdir" || E.cod || "asigna";
                }
```

Recupero el token_num de la TS (generado por el a. léxico la primera vez que encontró esta cadena) y la dirección de memoria reservada (generada por la parte del a.sintáctico que procesó la declaración de la variable ID).

Compongo en Ia.cod el código pila de la instrucción de asignación.

3. Generación de código para una máquina pila.

UNIVERSIDAD FRANCISCO DE VITORIA

Expresión aritmética.

DDS (Definición dirigida por sintaxis)

```
E → E + T      { E.cod = E2.cod || T.cod || "suma" }
  | E - T      { E.cod = E2.cod || T.cod || "resta" }
  | T          { E.cod = T.cod }
T → T * F      { T.cod = T2.cod || F.cod || "mult" }
  | T / F      { T.cod = T2.cod || F.cod || "div" }
  | F          { T.cod = F.cod }
F → - F        { F.cod = F2.cod || "inserta -1" || "mult" }
  | ID         { token_num = BuscaTokenEnTS(TS,ID);
                 memdir = BuscaAtribEnTS(TS,token_num,"MEMDIR");
                 F.cod = "valord memdir" }
  | NUM        { F.cod = "inserta NUM.valor" }
  | ( E )      { F.cod = E.cod }
```

3. Generación de código para una máquina pila.

UNIVERSIDAD FRANCISCO DE VITORIA

Expresión booleana

DDS (Definición dirigida por sintaxis)

```
Eb → Eb or Tb { Eb.cod = Eb2.cod || Tb.cod || "or" }
   | Tb       { Eb.cod = Tb.cod }
Tb → Tb and Fb{ Tb.cod = Tb2.cod || Fb.cod || "and" }
   | Fb       { Tb.cod = Fb.cod }
Fb → not Fb   { Fb.cod = Fb2.cod || "not" }
   | ID       { token_num = BuscaTokenEnTS(TS,ID);
                memdir = BuscaAtribEnTS(TS,token_num,"MEMDIR");
                Fb.cod = "valord memdir" }
   | true     { Fb.cod = "inserta 1" }
   | false    { Fb.cod = "inserta 0" }
   | ( Eb )   { Fb.cod = Eb.cod }
```

Añado "and", "or", y "not" al repertorio de instrucciones de la máquina pila abstracta.

Elijo la codificación interna "0" para "false" y "1" para "true".

3. Generación de código para una máquina pila.

UNIVERSIDAD FRANCISCO DE VITORIA

Expresión booleana (con evaluación perezosa)

DDS (Definición dirigida por sintaxis)

```
Eb → Eb or Tb { etiq=NuevaEtiqueta();
                Eb.cod = Eb2.cod || "copia" || "siciertovea etiq" ||
   Tb.cod || "or" || "etiqueta etiq" }
   | Tb       { Eb.cod = Tb.cod }
Tb → Tb and Fb{ etiq=NuevaEtiqueta();
                Tb.cod = Tb2.cod || "copia" || "sifalsovea etiq" || Fb.cod
   || "and" || "etiqueta etiq"}
   | Fb       { Tb.cod = Fb.cod }
Fb → not Fb   { Fb.cod = Fb2.cod || "not" }
   | ID       { token_num = BuscaTokenEnTS(TS,ID);
                memdir = BuscaAtribEnTS(TS,token_num,"MEMDIR");
                Fb.cod = "valord memdir" }
   | true     { Fb.cod = "inserta 1" }
   | false    { Fb.cod = "inserta 0" }
   | ( Eb )   { Fb.cod = Eb.cod }
```

Tipo de modificaciones de código que realiza un optimizador. Suelen mejorar el rendimiento sin afectar a la semántica.

Añado código a la evaluación del "and" y "or" para que:

-"and": si el primer operando ya es falso, salte la ejecución del código del segundo operando.
- "or": si el primer operando ya es cierto, salte la ejecución del código del segundo operando.

3. Generación de código para una máquina pila.

UNIVERSIDAD FRANCISCO DE VITORIA

Expresión de comparación

DDS (Definición dirigida por sintaxis)

Hemos añadido nuevas instrucciones a la máquina pila para poder realizar comparaciones aritméticas.

```
Ec → Eb                 { Ec.cod = Eb.cod }
   | Tc OpRe Tc         { Ec.cod = Tc1.cod || Tc2.cod || OpRe.cod }
Tc → E                  { Tc.cod = E.cod }
   | Eb                 { Tc.cod = Eb.cod }
   | ( Ec )             { Tc.cod = Ec.cod }
OpRe → <                { OpRe.cod = "comp_menor" }
   | >                  { OpRe.cod = "comp_mayor" }
   | ==                 { OpRe.cod = "comp_igual" }
   | !=                 { OpRe.cod = "comp_dist" }
   | <=                 { OpRe.cod = "comp_menorigual" }
   | >=                 { OpRe.cod = "comp_mayorigual" }
```

3. Generación de código para una máquina pila.

UNIVERSIDAD FRANCISCO DE VITORIA

Proposición *if-then*

DDS (Definición dirigida por sintaxis)

```
Iif → if Ec then L    { etiq=NuevaEtiqueta();
                        Iif.cod = Ec.cod || "sifalsovea etiq" || L.cod ||
    "etiqueta etiq"
                      }
```

3. Generación de código para una máquina pila.

UNIVERSIDAD FRANCISCO DE VITORIA

Proposición *if-then-else*

DDS (Definición dirigida por sintaxis)

```
Iif → if Ec then L Iif_2
                        { etiq_else=NuevaEtiqueta();
                          etiq_fin=NuevaEtiqueta();
                          Iif.cod = Ec.cod || "sifalsovea etiq_else" || L.cod
    || Iif_2.cod
                          Iif_2.etiq_else=etiq_else
                          Iif_2.etiq_fin=etiq_fin
                        }
Iif_2 → else L          { Iif_2.cod="vea Iif_2.etiq_fin" || "etiqueta
    Iif_2.etiq_else" || L.cod || "etiqueta Iif_2.etiq_fin" }
        | e             { "etiqueta Iif_2.etiq_else" }
```

3. Generación de código para una máquina pila.

UNIVERSIDAD FRANCISCO DE VITORIA

Proposición *while*

DDS (Definición dirigida por sintaxis)

```
Iw → while Ec do L   { etiq_inicio=NuevaEtiqueta();
                       etiq_fin=NuevaEtiqueta();
                       Iw.cod = "etiqueta etiq_inicio" || Ec.cod ||
   "sifalsovea etiq_fin" || L.cod || "vea etiq_inicio" || "etiqueta
   etiq_fin" }
```

3. Generación de código para una máquina pila.

UNIVERSIDAD FRANCISCO DE VITORIA

Proposición *for*

DDS (Definición dirigida por sintaxis)

```
Ifor → for ID := E to E do L
                { etiq_inicio=NuevaEtiqueta();
                  etiq_fin=NuevaEtiqueta();
                  token_num = BuscaTokenEnTS(TS,ID);
                  memdir = BuscaAtribEnTS(TS,token_num,"MEMDIR");
                  Ifor.cod = "valori memdir" || E1.cod || "asigna" ||
   "etiqueta etiq_inicio" || "valord memdir" || E2.cod ||
   "comp_menorigual" || sifalsovea etiq_fin" || L.cod || "valori memdir"
   || "valord memdir" || "inserta 1" || "suma" || "asigna" || "vea
   etiq_inicio" || "etiqueta etiq_fin" }
```