TABLA DE SÍMBOLOS

- **TABLA DE SÍMBOLOS**

  Estructura de datos que contiene un registro por cada identificador, con los campos para los atributos:

  -- Información sobre la memoria asignada

  -- tipo

  -- ámbito

  -- Si es nombre de procedimiento (número, tipo y método de paso de cada argumento)

- **Permite encontrar rápidamente cada ID y almacenar o consultar datos de ese registro**

- **En el Análisis Léxico se detectan los ID y se introducen en la Tabla de Símbolos**

- **Las fases restantes introducen información sobre los ID y después la utilizan**

TABLA DE SÍMBOLOS

- **Una tabla de símbolos puede servir para los siguientes propósitos dependiendo del lenguaje en cuestión:**
  - **Almacenar los nombres de todas las entidades de forma estructurada en un solo lugar.**
  - **Para verificar si se ha declarado una variable.**
  - **Para implementar la comprobación de tipos, verificando que las asignaciones y expresiones en el código fuente son semánticamente correctas.**
  - **Para determinar el alcance de un nombre (resolución de alcance).**
- **Una tabla de símbolos es simplemente una tabla que puede ser lineal o una tabla hash. Mantiene una entrada para cada nombre en el siguiente formato:**

<symbol name, type, attribute>

static int interest; <interest, int, static>

IMPLEMENTACIÓN TABLA DE SÍMBOLOS

- **Una tabla de símbolos puede implementarse de una de las siguientes maneras:**
  - **Lista lineal (ordenada o sin ordenar)**
  - **Árbol de búsqueda binario**
  - **Tabla Hash**

- **Las tablas de símbolos se implementan mayoritariamente como tablas hash,**
- **donde el propio símbolo del código fuente se trata como una clave para la función hash y el valor de retorno es la información sobre el símbolo.**

```
using System.Collections;

                    // Create a new hash table. //
                    Hashtable ht = new Hashtable();
```

OPERACIONES TABLA DE SÍMBOLOS

- Una tabla de símbolos, ya sea lineal o hash, debe proporcionar las siguientes operaciones.
  - insert()
  - lookup()

- La operación insert() se utiliza en la etapa de análisis, donde los tokens son identificados y los nombres son almacenados en la tabla

- Un atributo para un símbolo en el código fuente es la información asociada a ese símbolo. Esta información contiene el valor, el estado, el ámbito y el tipo del símbolo.

- La función insert() toma el símbolo y sus atributos como argumentos y almacena la información en la tabla de símbolos.

```
int a;
```

```
insert(a, int);
```

OPERACIONES TABLA DE SÍMBOLOS

- La operación lookup() se utiliza para buscar un nombre en la tabla de símbolos para determinar:
  - si el símbolo existe en la tabla.
  - si se declara antes de ser utilizado.
  - si el nombre se utiliza en el ámbito.
  - si el símbolo está inicializado.
  - si el símbolo se declara varias veces.
- El formato de la función lookup() varía según el lenguaje de programación. El formato básico debe coincidir con el siguiente:

```
lookup(symbol)
```

- Este método devuelve 0 (cero) si el símbolo no existe en la tabla de símbolos. Si el símbolo existe en la tabla de símbolos, devuelve sus atributos almacenados en la tabla.

C# Copy

```
using System;
using System.Collections;

class Example
{
    public static void Main()
    {
        // Create a new hash table.
        //
        Hashtable openWith = new Hashtable();

        // Add some elements to the hash table. There are no
        // duplicate keys, but some of the values are duplicates.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // The Add method throws an exception if the new key is
        // already in the hash table.
        try
        {
            openWith.Add("txt", "winword.exe");
        }
        catch
        {
            Console.WriteLine("An element with Key = \"txt\" already
```

No permite duplicados

DETECCIÓN E INFORMACIÓN DE ERRORES

- **Cada fase puede encontrar errores y debe tratarlo para continuar con la Compilación, permitiendo detectar más errores**

- **Las fases de Análisis Léxico, Sintáctico y Semántico manejan la mayoría de los errores**

Pero ¿Qué detectamos en el análisis semántico?

- **En el Análisis Semántico se detectan errores donde la estructura sintáctica es correcta pero no tiene significado la operación**

  (Por. ej. sumar dos ID , donde uno es el nombre de una matriz y el otro un nombre de procedimiento)

GENERACIÓN DE CÓDIGO INTERMEDIO

- Se genera una representación intermedia explícita del PF
- La representación intermedia es como un programa para una máquina abstracta
- Esta representación debe ser fácil de producir y de traducir al programa objeto
- Una de ellas es el *"código intermedio de 3 direcciones"*
- Ejemplo

```
t1 := entareal (60)
t2 :=  id3 + t1
t3 := id2  + t2
id1 := t3
```

La Optimización es problemática

```
for (i = 0; i < N; i += 1)
    A[i] *= D/A[0];
```

is NOT

```
tmp1 = D/A[0];
for (i = 0; i < N; i += 1)
    A[i] *= tmp1;
```

OPTIMIZACIÓN DE CÓDIGO

- Trata de mejorar el código intermedio para que resulte un código de máquina más rápido de ejecutar

- En el ejemplo: t1 : = id3 * 60.0
  id1 : = id2 + t1
  La conversión a real se hace en compilación

- *Compiladores optimizadores* : La fase de optimización ocupa una parte significativa del tiempo del compilador

- Hay optimizaciones sencillas que mejoran el tiempo de ejecución del programa sin retardar mucho la compilación

Loop Unrolling

Source:

```
for i := 1 to 100 by 1
  A[i] := A[i] + B[i];
endfor
```

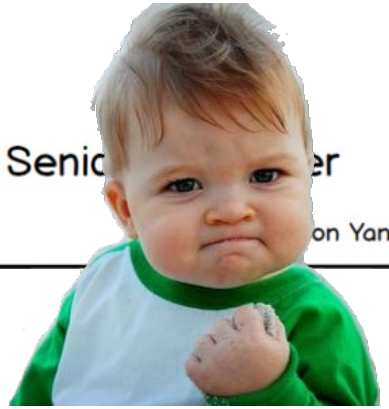
Transformed Code:

```
for i := 1 to 100 by 4
  A[i  ] := A[i  ] + B[i  ];
  A[i+1] := A[i+1] + B[i+1];
  A[i+2] := A[i+2] + B[i+2];
  A[i+3] := A[i+3] + B[i+3];
endfor
```

- ¿Cómo medir rendimiento?

printf("Hello World"); printf("Hello World"); printf("Hello World"); printf("Hello World"); printf("Hello World");	for (int i=0; i<5; i++) printf("Hello World");	printf("Hello World"); printf("Hello World"); printf("Hello World"); printf("Hello World"); printf("Hello World");
CS Student	Junior Programer	Senic er

on Yang



GENERACIÓN DE CÓDIGO

- La fase final genera *código objeto* (en general código de máquina relocalizable o código ensamblador)
- Se seleccionan las posiciones de memoria para las variables usadas por el programa.
- Se traduce cada una de las instrucciones intermedias a una secuencia de instrucciones de máquina
- Un aspecto decisivo es la asignación de variables a registros.
- En el ejemplo, utilizando los registros 1 y 2:

```
MOVF    id3, R2
MULF   %  60.0, R2
MOVF    id2, R1
ADDF    R2, R1
MOVF    R1, id1
```

PROGRAMAS RELACIONADOS CON UN COMPILADOR

- **PREPROCESADORES** (producen la entrada para un comp.)
  - Procesamiento de Macros
  - Inclusión de archivos
  - Preprocesadores " racionales" (*estruct. de control)*
  - Extensiones a lenguajes (*bases de datos)*

- **ENSAMBLADORES**

  Producen código ensamblador que se pasa a un ensamblador para su procesamiento (versión mnemotécnica del código de máquina: nombres de operaciones y nombres de direcciones de memoria)

- **ENSAMBLADO DE DOS PASADAS** (lecturas del archivo IN)
  - *Primera:* Identificadores - Tabla de símbolos
  - *Segunda*: Traduce códigos de operaciones e identificadores
    El resultado es código de maquina relocalizable

- **CARGADORES Y EDITORES DE ENLACE**

  Modifica las direcciones relocalizables y ubica en memoria.
  Forma un solo prog. desde varios archivos relocalizables

AGRUPAMIENTO DE FASES EN LA IMPLEMENTACION

ETAPA INICIAL Y ETAPA FINAL

Inicial : **Fases que dependen del lenguaje fuente**
Hasta cierta optimización

Final : **Partes que dependen de la maq. objeto y del leng. intermedio**

PASADAS

Se agrupan las actividades de varias fases en una misma pasada (lectura de un archivo de entrada y escritura de un archivo de salida)

REDUCCION DEL NUMERO DE PASADAS

Pocas pasadas --> Varias fases dentro de una pasada --> Prog. completo en memoria en representación intermedia Fusión de código intermedio y objeto: " backpatching"

```
from random import randint

numero_secreto = randint( 1, 10 )
intentos = 3

while True:
```

```
numero_secreto = mate.aleatorio( 1, 10 )
intentos = 3

mientras verdadero

    escribir( "Adivina el número (1-10)>>> " )
    elige_numero = leer()

    elige_numero = anumero( elige_numero )

    si( intentos < 1 )

        escribir( "\n Demasiados intentos, has perdido :(" )
        sis.salir()

    osi( elige_numero == numero_secreto )

        escribir( "\n ¡¡Has acertado!!" )
        sis.salir()

    sino

        escribir( "\n Fallaste, te quedan "..intentos.." intentos. ¡Prueba otra vez!" )
        intentos -= 1

    fin

fin
```

HERRAMIENTAS PARA CONSTRUCCIÓN DE COMPIL.

- **SIST. DE AYUDA PARA ESCRIBIR COMPILADORES**

  Compilador de comp. / Generadores de comp. /

  Sist. generadores de traductores

- **HERRAMIENTAS GENERALES PARA EL DISEÑO AUTOMÁ-TICO DE COMPONENTES ESPECÍFICOS DE UN COMP.**

  Utilizan leng. específicos para especificar e imple-mentar la componente

  Ocultan detalles del algoritmo de generación

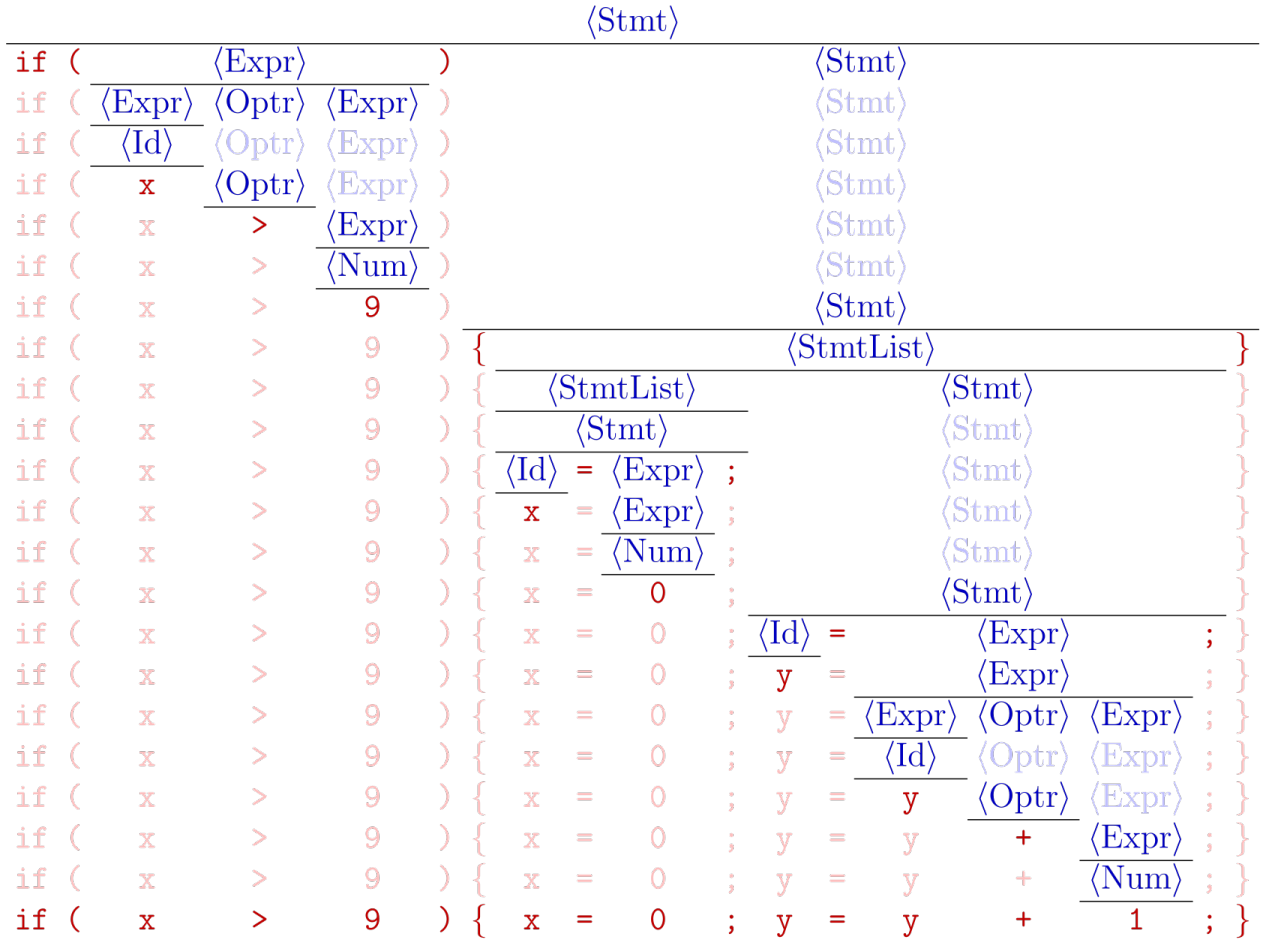
  Producen componentes que se pueden integrar al resto del compilador

HERRAMIENTAS PARA CONSTRUCCIÓN DE COMPIL.

GENERADORES DE ANALIZADORES SINTACTICOS

Producen AS a partir de una Gramática Libre de Contexto
Hoy esta es una de la fases más fáciles de aplicar

⟨Stmt⟩ → ⟨Id⟩ = ⟨Expr⟩ ;
⟨Stmt⟩ → { ⟨StmtList⟩ }
⟨Stmt⟩ → if (⟨Expr⟩) ⟨Stmt⟩
⟨StmtList⟩ → ⟨Stmt⟩
⟨StmtList⟩ → ⟨StmtList⟩ ⟨Stmt⟩
⟨Expr⟩ → ⟨Id⟩
⟨Expr⟩ → ⟨Num⟩
⟨Expr⟩ → ⟨Expr⟩ ⟨Optr⟩ ⟨Expr⟩
⟨Id⟩ → x
⟨Id⟩ → y
⟨Num⟩ → 0
⟨Num⟩ → 1
⟨Num⟩ → 9
⟨Optr⟩ → >
⟨Optr⟩ → +



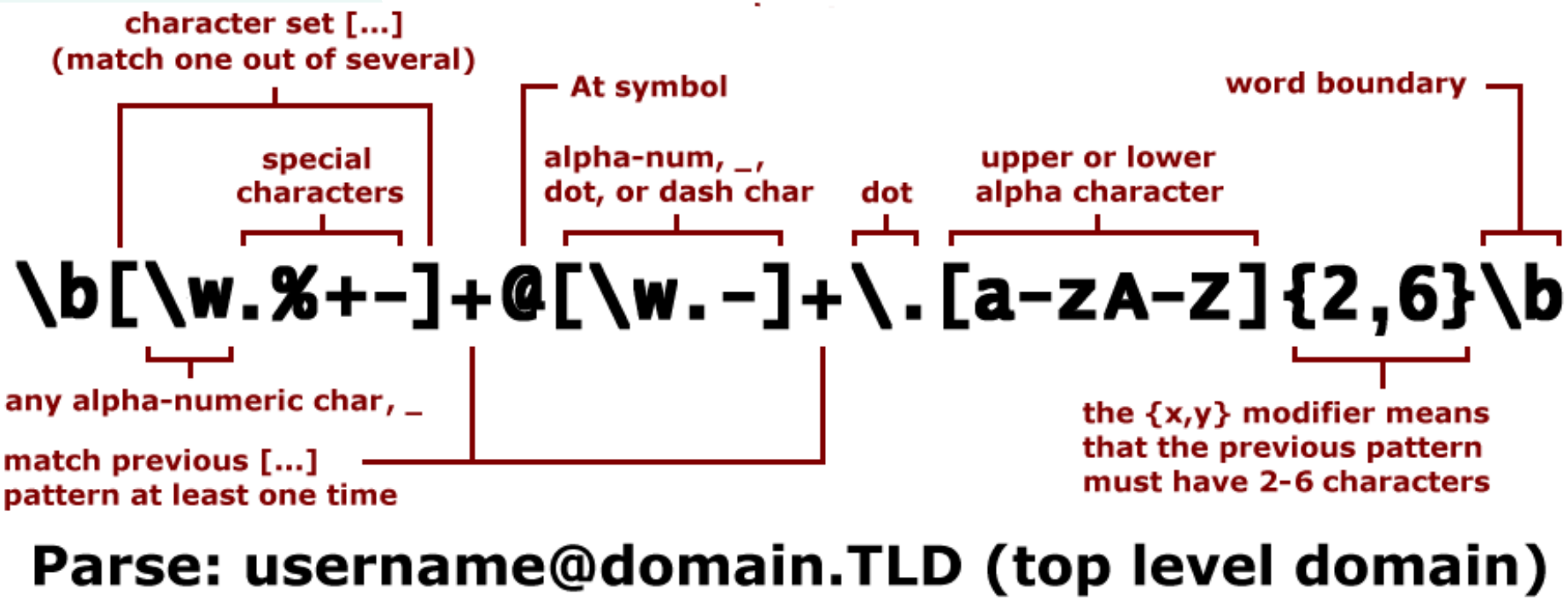
HERRAMIENTAS PARA CONSTRUCCIÓN DE COMPIL.

- **GENERADORES DE ANALIZADORES SINTACTICOS**

  Producen AS a partir de una Gramática Libre de Contexto
  Hoy esta es una de la fases más fáciles de aplicar

- **GENERADORES DE ANALIZADORES LEXICOS**

  Producen AL a partir de una especificación en Expres. Regulares. El AL resultante es un Autómata Finito



HERRAMIENTAS PARA CONSTRUCCIÓN DE COMPIL.

- **GENERADORES DE ANALIZADORES SINTACTICOS**

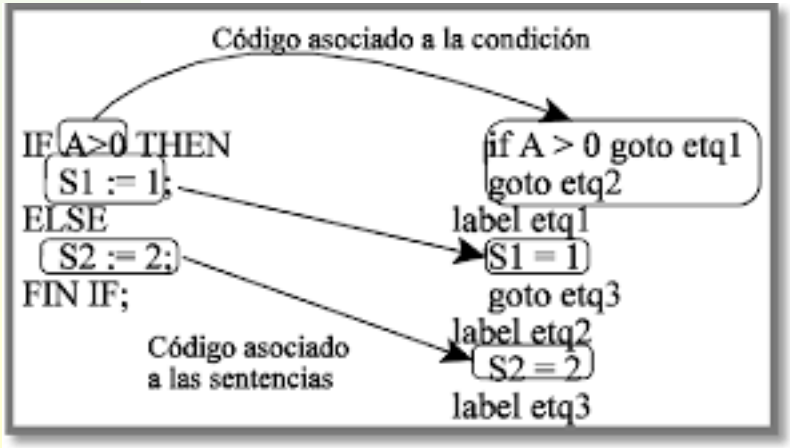
  Producen AS a partir de una Gramática Libre de Contexto
  Hoy esta es una de la fases más fáciles de aplicar

- **GENERADORES DE ANALIZADORES LEXICOS**

  Producen AL a partir de una especificación en Expres. Regulares. El AL resultante es un Autómata Finito

- **DISPOSITIVOS DE TRADUC. DIRIGIDA POR LA SINTAXIS**

  Producen grupos de rutinas que recorren el árbol de AS generando código intermedio



HERRAMIENTAS PARA CONSTRUCCIÓN DE COMPIL.

- **GENERADORES DE ANALIZADORES SINTACTICOS**

  Producen AS a partir de una Gramática Libre de Contexto
  Hoy esta es una de la fases más fáciles de aplicar

- **GENERADORES DE ANALIZADORES LEXICOS**

  Producen AL a partir de una especificación en Expres. Regulares. El AL resultante es un Autómata Finito

- **DISPOSITIVOS DE TRADUC. DIRIGIDA POR LA SINTAXIS**

  Producen grupos de rutinas que recorren el árbol de AS generando código intermedio

- **GENERADORES AUTOMÁTICOS DE CÓDIGO**

  Las proposiciones en cod. Int. se reemplazan por plan-tillas que representan secuencia de instruc. de máquina

- **DISPOSITIVOS PARA ANALISIS DE FLUJO DE DATOS**

  Inf. sobre como los valores se transmiten de una parte a otra del programa

HERRAMIENTAS PARA CONSTRUCCIÓN DE COMPIL.

Lex y YACC

- Herramientas que nos permiten desarrollar componentes o la mayor parte de un compilador
- Son un recurso invaluable para el profesional y el investigador
- Existen paquetes freeware
- ANTLR

ALGUNOS TIPOS ESPECIALES DE COMPILADORES

- **COMPILE- LINK- GO**
  Se compilan segmentos por separado y luego se montan todos los objetos producidos en un módulo cargable listo

- **COMPILADOR DE VARIAS PASADAS**
  No es más lento. Ocupa poca memoria. Fácil de mantener

- **COMPILADOR INCREMENTAL (o interactivo)**
  Se pueden compilar solo las modificaciones

- **AUTOCOMPILADOR**
  Comp. escrito en el propio leng. que traduce. Portabilidad.

- **METACOMPILADOR**
  Programa al que se le especifica el lenguaje para el que se quiere un comp. y produce el comp. como resultado

- **DECOMPILADOR**
  Traduce de código máquina a leng. de alto nivel

EL LENGUAJE Y LA HERRAMIENTA

MODELO	LENGUAJE	CARACTERISTICAS
Compilado	Fortran, COBOL, C/C++, Pascal	Sintaxis rigurosa, velocidad y tamaño
Interpretado	Lisp,AWK, BASIC, SQL	Desempeño lento. Actividades no planea-das. Sintaxis relajadas
Pseudocompilado	Java	Transportabilidad absoluta, desempeño intermedio. Sintaxis rigurosa

ASPECTOS ACADEMICOS Y DE INVESTIGACION

AREA	BENEFICIOS
Lenguaje de prog.	Principios para su desarrollo Herramientas para implementación
Inteligencia artificial	Interfases de reconocimiento de lenguaje natural
Sistemas operativos	Desarrollo de interfases de control y usuario final. Intérpretes de comandos (shells)
Diseño de interfaces	Desarrollo de interf. orientadas a comando y carácter. Voz o escritura
Administración de proyectos inform.	Selección de herramientas de desarrollo. Evaluación de costo y beneficios.