**1 GENERACION**

**Lenguaje maquina:** binario, código de instrucción, operando normalmente

**Lenguaje ensamblador**: en vez de escribirlo nosotros, a cada parte le asigna un nombre, es decir, un símbolo. En este lenguaje se le asigna un valor a cada instrucción en código binario. Por ejemplo, ponemos “load” y en una tabla busca “load” y le pone el código correspondiente a “load”. Se traduce a la memoria la parte del operando (variables) introduciéndolos en una tabla (tabla de símbolos) con una serie de restricción

Por ejemplo:

Nombre-memoria

Pepe – 0

Juan - 1

Luego la traducción del lenguaje depende de dos tablas; tabla símbolos para instrucciones como “load” y una tabla de símbolos ( que depende del programa)

**2** **GENERACION**

**Macroensamblador:** hay determinadas secuencias de instrucciones que se van a repetir, y para no repetirlas, se empaquetan y se hace una macro. un macro ensamblador tendrá un repertorio y una serie de macros con su nombre y los valores que correspondan. Una macro te da una lista de instrucciones que desarrolla.

**3 GENERACION**

**Fortran:** es el primer lenguaje para el que se hace un compilador eficiente. Su compilador era muy eficiente.

**Lenguajes de alto nivel:** era un lenguaje de símbolos e incorporaba estructuras en los datos y en las instrucciones. El paso de un ensamblador a un macroensamblador era un hilo de estructuras. Este lenguaje te permite crear sentencias de alto de nivel que permite manejar instrucciones y permiten la estructura de datos.

Si hago algo a complemento a uno, estoy sumando y se me desborda un uno, el elemento que arrastro lo tengo que volver a sumar. En el complemento a uno, una suma puede acarrear una suma. En el complemento a dos, si arrastro un numero lo elimino y una suma solo tiene una suma. // Este párrafo tiene sentido aunque parezca que no, pensar…

//Los números reales como 478.25 se guarda como 0.47825\*10. Todos los números reales empezaran por 01 y se elevan a 2^0. Para un número real se almacena: el signo, el número y el exponente. Pues el formato es signo-exponente-número y el numero siempre tiene que empezar por 1. //

Las estructuras de datos son arrays, registros que se agrupan en ficheros….

Posteriormente aparece el **concepto de tipo de dato**: que se dividen en **primitivos** y **estructurados**:

**-Los tipos primitivos:** son los que primitivamente se manejaban, el entero, el natural, tipo char. Y ya vienen definidos en el lenguaje

-**Tipos estructurados**: so los que tienen alguna estructura en base a tipos primitivos o en base a tipos estructurado.

Un **tipo** se define como una terna G O P: donde g es el conjunto de valores que se puede tomar dentro del tipo. O, son las operaciones que soporta el tipo, P son las propiedades que tienen esas operaciones.

*Ejemplo:* el tipo entero el género sería desde -2^31 a 2^31. Las operaciones serían

Suma resta, multiplicación y división. Las propiedades son: los axiomas.

En Los tipos estructurados, lo que conoce el lenguaje es la estructura del tipo pero no conoce el género que lo compone ¨Por ejemplo, una matriz se estructura en filas y en columnas, si definimos un arrays dando el número de filas y columnas no se tiene establecido el tipo de dato, luego tendré que dar lo que esté dentro.¨ Pues para definir un array, sabiendo que es una estructura de tabla, tenemos que dar el número de filas y de columnas y lo que hay dentro.

// Como conclusión de lo dicho anteriormente

En los tipos primitivos el lenguaje lo conoce todo porque no varía nada. En los tipos estructurados, conoce macroscópicamente la estructura pero hay que decir las filas, columnas y lo que hay dentro. //

Posteriormente se incorporaron los **tipos de datos abstractos** que son tipos de datos que el lenguaje no conoce ni la estructura, ni las propiedades ni las ordenaciones. Pero se puede definir si el programador hace la implementación del tipo de dato.

En las instrucciones se incorpora estructura, pero ¿a qué? :

-Expresiones

-Control de flujo: como instrucciones del tipo if e inducciones del tipo bu. La ida del dato incondicional se pasó al if y se mezclaron con las expresiones.

Desde el punto de vista de la traducción con la estructura del control de flujo aparecen dos tipos de sentencias:

-**Sentencia semánticamente simple**: su significado solo depende de ella y de nada más. Por ejemplo, una asignación. Por tanto, las SSS son las sentencias asignación (variable=3+5) y las operaciones de entrada y salida de una sola variable (una sentencia es una orden)

-**Sentencia semánticamente compuestas**: el resto

Si estamos en ensamblador no hay SSC, todas las sentencias son iguales. Entonces cuando ya tengo el salto de ensamblador a macroensamblador aparece la idea de SSC y cuando demos el salto a los lenguajes de alto nivel aparecen SSS y SSC.

**9-11-2017**

**LENGUAJE MAQUINA**

**ENSAMBLADOR**

**MACRO ENSABMBLADOR**

**LENGUAJE ALTO NIVEL:**

**-DATOS: (TIPOS)-> G OP P**

**-PRIMITIVO**

**-ESTRUCTURADO**

**-TIPO DATOS ABSTRACTOS**

**-SENTENCIAS:**

**-EXPRESIONES**

**-SENTENCIAS COMPUESTAS**

La estructura del género G tiene que ser determinada. Por ejemplo, un vector es del tipo estructurado,

Supongamos un vector (2., 3.1, 4.2) cual sería el género del tipo al que pertenece el vector? :

Si el género son los números reales significa que es primitivo, podría ser también R3, pero no lo es porque el tipo de dato no te permite multiplicarlo como una matriz. Es decir, en matemáticas podemos hacer operaciones con vectores-matrices pero en informática no se puede si son arrays. El tipo array de los lenguajes de programación, las operaciones que tienen definidas no son las de las matrices. Las matrices como tales en los lenguajes de programación no se pueden manejar porque sus operaciones no están definidas, eso lo tiene que hacer el programador. El género del tipo del vector anterior va de un conjunto que tenga 3 elementos. (0,1.2) y ese conjunto es una aplicación que a cada número del conjunto de los elementos le asigna un valor del vector, el género es R .

Los arrays no son matrices, lo único que tienen con ella son las estructuras espaciales de datos. Si el array es de los números reales, las operaciones son con reales, si fuese con enteros, las operaciones serán con enteros. Él género se construye con operaciones de conjuntos, sea el producto cartesiano….

**IMPORTANTE**: El tipo establece la representación de los valores de ese tipo. A los valores del vector se le asignan una dirección de memoria y a partir de ésta, reserva el equivalente a 3 números reales. Supongamos una matriz de tipo entero 3x4 y la matriz se llama Juana, A un valor de ese tipo se le asigna una dirección y a partir de esa dirección 12 posiciones de números enteros.

Cuando en un array con n valores, buscamos el valor cero, accede a la primera posición, por tanto, el tamaño va de 0 a n-1. Así que, si le damos el valor n para que lo lea o para asignarle algún valor, lo estaremos haciendo mal. Es importante comprobar que el array está en el rango en el que deba estar.

**Sentencias:** los lenguajes de alto nivel incorporan expresiones (d+f\*f….) y por otra parte incorporan sentencias compuestas que inciden en la **ruptura de flujo,.**

Con las sentencias SSC,

Cuando a=b, el lenguaje máquina carga b, me lo llevo a un registro y lo almaceno en a. Si en vez de una variable tengo una expresión, el proceso es lo mismo, pero toda la parte de la expresión se descomponen para obtener un valor de la misma y esto lo hace operación a operación. Si tengo una instrucción de leer a b c d, el ensamblador las recompone una a una y las muestra.

*Ejemplo.*

Si tengo un if expresión

El patrón lo que hace es evaluar la expresión y dependiendo de si es verdad o no la ejecuta. Es el mismo patrón que desarrolla un bucle definido como el for: (que tiene 3 expresiones, inicial, incremento y final). La semántica del bucle es ejecutar el cuerpo tal número de veces y ese número de veces está determinado antes de ejecutarlo. Ese número se obtiene expresión final menos inicial entre incremento.

**15-11-2017**

El de la cocina se ha cambiado a profesor de lengua.

Si vamos a construir un lenguaje, lo lógico es que el lenguaje sea conciso, concreto, que nos sea ambiguo. Un lenguaje se define:

-Vocabulario: conjunto de palabras con significado

-Gramática: definir una estructura y el vocabulario forma parte de la gramática. La estructura que se define es sintáctica. Lo que permite la gramática es escribir frases correctas con palabras correctas. Lo que define a la gramática es la sintaxis y es la forma de construir, no tiene nada de significado.

En resumen, en un lenguaje hay que tener una serie de palabas que vienen del diccionario y tienen asociadas un significado atómico, de tal forma que el significado de una palabra no depende del significado de otra palabra. Pues todo esto lo aplicamos aun lenguaje de programación o un lenguaje artificial construido por nosotros. El lenguaje en principio es solo forma, solo sintaxis.

Una gramática se define con una cuadrupla **G = (Vn, Vt, P, S)**

**Vn**: es el conjunto de símbolos no terminales

**Vt**: es el conjunto de símbolos terminales

**P:** es el conjunto de producciones

**S:** es el símbolo inicial

Los **signos terminales** son las palabras y los **signos no terminales** son aquellos símbolos construidos con palabras. Por ejemplo los prefijos serían símbolos no terminales y las palabras terminales.

La ambigüedad sintáctica es que cada elemento no terminal tenga una sola estructura. Esas estructuras tienen que tener forma de árbol. Pues si tiene estructura de árbol, cuando aplique la gramática todos van a llevar a un elemento raíz, pues a eso lo consideramos el **símbolo inicial**, S, que será un elemento terminal que los engloba a todos. Si en lugar de estar recorriendo el árbol hacia arriba lo que hacemos es generar frase, nos damos cuenta de que a partir del símbolo inicial aplicando distintas reglas podremos generar todo el árbol.

Ejemplo de las transparencias:

Símbolo terminal->estructura del símbolo no terminal

Un símbolo puede tener más de una estructura, cuando hay más alternativas se usa una barra vertical. La E y la O tienen una estructura. Una O puede ser un + - \* ó /. Cada cosa tendrá un significado y en la expresión dice que puede ser una E operada con otra E ó puede ser una expresión entre paréntesis o puede ser un id. El id es un símbolo terminal y es un identificador, es decir, un nombre de lo que sea.

Una posible estructura es la que hay en la derecha de la imagen. Añadimos una regla semántica de evaluación que es la expresión del id.

**16/11/2017**

Algunas estructuras se nos pueden quedar ambiguas como en el ejemplo del folio que tengo que tener por ahí, para evitar esa ambigüedad hay que darle por ejemplo prioridad a los paréntesis. Con el if pasa algo similar y para evitar esa ambigüedad hay que ponerlo con llaves

Una estructura se define como G= (N, T, P, S)

**…….**

Con el paso del tiempo se construyeron autómatas que reconocieran este tipo de gramática. Dentro de esos autómatas las expresiones regulares se conocen como autómatas finitos. La gramática independiente del concepto se llama AUTOMATA CON PILA, la pila se usa para ir almacenando para sacar la información que el autómata reconoce.

Desde el punto de vista del lengua de programación, las G. Regulares permiten identificar el léxico, la G.I.C permite identificar la sintaxis. Entre GIC y GSC, hay dos tipos de gramáticas, gramática con atributos y gramática con dos niveles, y la segunda tiene un poca más de potencia expresiva que la primera. La gramática por atributos se utiliza para especificar la semántica de ellos lenguajes de programación. La G. por atributos es una cuádrupla como la ya mencionada anteriormente.

Como se contruye una G por Atributos Para especificar la semántica del lenguaje? La Gpor atribtos es una cuadrupla Ga= ( Gic, A atibutos, E reglas de evaluación de atributos, e (condiciones sobre los valores de atributos. ))

Para definir la gramática de los enteros

Digito: 0123456…

Constante entera está formada por un digito o a una constante + un digito

Luego hay que asignarle algún valor

**17/11/2017**

**22/11/2017**

<letra> a b c …..z

<digito> 1,2,3,4,5,6,7,8,9

Letradigito-><letra> | <digito>

<identificador> -> <letra> | <identificador> <letradigito>

Todos los caracteres que estén en el texto tienen tb que estar en la gramática

La estructura sintáctica se representa en torno de token. Un token es la abstracción de lo particular a lo general. El nivel más bajo de la sintáxis es el nivel de tokens.

Por ejemplo, si tenemos juan:=pepe+10;

El compilador va leyendo caracteres y va comprobando con la gramática que tenemos arriba. Si tuviésemos ju?an:=pepe+10, cuando llega a la interrogación el compilador te dice que eso está mal y sigue leyendo el resto de la sentencia. El término “:=” es un separador y se pone porque si no podría ser (juan pepe) un único identificador.

Si estamos usando librerías cuando aparezca el nombre de una función lo que le dices al compilador es que busque dicha función en la librería. Cuando en la parte de declaraciones lee enteros (juan), como está en la parte de arriba busca si ya hay un juan y si no está lo apunta en la tabla de lexema (porque la estamos inicializando)

**Tabla de simbolos:**

Lexema tipo longitud(b) dirección valor

Juan int 2 0

P Real 4 2

L int 2 6 // Es 6 porque la anterior variable tiene 4 bits mas dos de la primera son 6 npor tanto ésta tiene la dirección 6//

De esta forma va reconociendo los nombres. Un compilador de un lenguaje normal con declaraciones explícitas de variable (en la parte de declaraciones tienes que decir el tipo). (Las declaraciones implícitas las veremos mejor con un ejemplo, si ponemos 10, el compilador lo lee como un entero y si ponemos 10. el compilador lo lee como coma flotante, eso son las declaraciones implícitas.) que reconoce un lexema se le da un valor y se escribe en la tabla de símbolos.

TODO ESTO ES A NIVEL LÉXICO.

Tras lo del léxico el compilador sigue leyendo, tras leer juan (identificador,) lee pepe (otro iden), lee el + que es un operador y luego lee el 10 que es un digito….

Las funciones void actúan como si fuese un procedimiento y no una función, no devuelve ningún tipo de dato.

El léxico cuando reconoce un lexema, al sintáctico le pasa el tokens. El analizador sintáctico lo que espera es una sentencia de asignación.

Los lenguajes de programación están hechos de tal manera que cuando le pasas una sentencia éste la reconoce y ya sabe cómo es. Los siguientes tokens que le va mandando el sintáctico, lo que hace es ir encajando las piezas para ver si la estructura de la frase es o no correcta. En el caso del ejemplo anterior, el de juan, lo que manda es un identificador, luego otro….y lo que se espera es una expresión.

Así sería el análisis sintáctico, pues según se van mandando los tokens, el sintáctico va creando el árbol.

Hay dos grandes familias de gramáticas:

-las que hacen el árbol sintáctico de forma descendente y de izquierda a derecha, es decir, lo primero que hacen es poner el axioma y a patir de ahí van construyendo hacia abajo el árbol sintáctico. Esta gran familia de gramáticas se conoce con el nombre de LL

-Hay otras familias más potentes LR, el autómata funciona de manera más complicada. El árbol es ascendente y en principio de derecha a izquierda

Entero pepe, juan

Leer juan

Pepe:= juan +10

Escribir pepe

Fin

Que es lo que haría haciendo un traductor con ese programa. Empezaría a leer , vería el léxico lo pasaría a la tabla de símbolos etc…

El sintáctico, en principio cuando se la manda un tokens es buscar que frase o tipo de frase empieza por un identificador y espera buscar una sentencia de asignación. Sigue leyendo el léxico y encuentra a juan y manda un identificador, ve que eso es una expresión, sigue avanzando, reconoce el operador pero no hace nada y sigue avanzado y llega a la constante 10, retrocede y construye el árbol sintáctico. El reconocimiento lo hace de forma ascendente .

Si ya tiene una estructura el sintáctico llama al semántico para ver si esa estructura tiene significado o no tiene significado porque puede ser sintácticamente correcta pero no semánticamente. Pepe + ‘10’ esa expresión no es correcta porque los tipos no son correctos ya que ‘10’ es un char, mientras que en la otra dice que el valor que tienen pepe se le suma 10 y pasa a ser el valor de juan. Si la operación de tipo es correcta lo siguiente que hace es ver si el tipo entero soporta la operación suma y luego si el entero puede ser operado con la siguiente varíale. En el ejemplo anterior, los enteros soportan la suma pero no se puede sumar un entero con un char y si fuera un entero con un real, el entero puede soportar la suma, y cuando se suma con el real habría que definir el tipo del resultado, en este caso como es un entero y un real, luego el resultado es real. Una operación de entero con entero lo que se evalua es el tipo de esa operación. Si la operación fuese muy larga, se iria evaluando esa expresión de tipos hasta que al final saliese el tipo.

Cuando se tiene que operar con expresiones complejas como a +b +c +(f+g) lo que se hace es guardar el resultado de la expreison entre paréntesis en una variable inventada y va operando con esa variable.

La mayoría de compiladores son compiladores dirigidos por sintaxis. Ademas de permitir expresiones complejas, los lenguajes de alto nvel permiten sentencias compuestas complejas de las cuales cada una tiene su significado y su patrón donde tienen un principio y un fin (if for…). Analiza toda la sentencia y genera el código correspondiente…..

**30/11/2017**

Definicion de intérprete: …

Formas de traducir:

-Interpretes

-Compiladores

Diferencia: el intérprete lo hace sentencia a sentencia, orden a orden… y el compilador coge todo el texto del programa, lo traduce, haciendo optimización etcetc y cuando lo tiene todo traducido nos da un fichero con un programa semánticamente equivalente mientras que el intérprete no entrega nada. El interprete coge una sentencia la traduce, la carga en la máquina y la ejecuta.

El lenguaje que tienen las maquinas como el móvil etc es interpretado ya que cada vez que le das una orden la traduce y la ejecuta, por ejemplo el lenguaje de control de un SO. Un móvil no puede ser traducido por compilador porque no se van a guardar todas las ordenes que le mandamos para ser traducidas toda de una vez.

Al interprete van comandos sentencias a sentencia las traduce a lenguaje maquina y las ejecuta

Y el compilador las traduce todas, no ejecuta nada y nos manda un archivos con la traducción.

Si todo nuestro programa es un bucle el compilador y el interprete es lo mismo, no habría diferencia, excepto que el interprete no daría la traducción. Un interprete en un bucle lo coge entero y lo ejecuta entero.

Los interpretes hacen menos optimizaciones que los compiladores. Los interpretes van mas rapidos ene l proceso de traducción pero son mas lentos en la traducción ya que el código que generan es menos eficiente porque no optimizan.

Un entorno de desarrollo integra en una herramienta el editor, interprete y compilador. Después de editar llega el intérprete, si hay algún error nos devuelve al texto y nos selecciona donde esta el fallo subrayado como en C++. Cuando ya funciona bien, empeiza a compilar, genera el código optimizado y guarda el otro.

**Otra cosa:¡ç**

Con lo que entrega el compilador se hace un mapa, el compilador genera el código maquina y en consecuencia sabe cuantos bytes genera. La información que entrega es código maquina, (todo lo que ha escrito), el mapa de memoria que él ha hecho a partir del código maquina y la tabla de simbolos donde especifica el tamaño del código y de los

La recursividad de funciones es cuando una función se llama a sí misma. Cuando se vuelva a llamar tendrá que crear una copia de sus valores y para ello necesita la tabla de símboos.

Cuando termina el compilador en la tabla de símbolos habrá referencias externas porque en el texto que le hemos dado para que traduzcan no están. Eso se queda ahí y eso es lo que entrega el compilador. El siguiente paso para ejecutar el programa es pasarlo por el enlazador o el encuadernador. El enlazador lo que hace es unir los distintos bloques y juntarlo en un solo bloque y el encuadernador resuelve todas esas referencias externas que antes el compilador no ha encontrado. Generalmente las busca en alguna biblioteca, cuando se resuelven todas las referencias le va dando las direcciones oportunas y cuando termina el proceso de encuadernación no debe quedar ninguna referencia sin dirección. El encuadernadoz compacta el código y los datos pero por separado y diseña el mapa de memoria total que va a tener el bloque entero cuando se está ejecutnado

El mapa que se construye con la información que entrega el encuadernador está diseñado con una dirección lógica 0 y de esa esa dirección 0 en adelante hasta la M pone el código. Las siguientes direcciones más bajas pone lo que son datos estáticos, que son datos cuyo tamaño no va a cambiar durante la ejecución del programa. Los datos estáticos son las constantes o las variables normalitas que usamos. El resto de espacio lo deja para los datos dinámicos que son los datos que van a cambiar durante la ejecución del programa. Esos datos dinámios con estructura de pila va creciendo hacia abajo y lo que deja es una pila donde cada elemento de la pila es lo que se llama un registro de activación.

Ese registro de activación tiene por una parte al principio y al final unos enlaces con la dirección del registro anterior y en el otro extremo con una dirección del siguiente. Cada registro se corresponde con una llamada a función. Al principio tiene una dirección donde se quedó el programa que llamo a esa función. Pues cuando se llama a una función se crea el registro de activación y el regisro para poder volver y continuar tiene que tener la dirección desde donde se ha llamado. Pues en definitiva, en la cabecera y en la cola tiene enlaces, y en medio tienen datos que utiliza la función. Entonces empieza con los argumentos que tenga la función pepe (int a int b) la a y b estaría en esa zona. Todo lo que sigue son las variables que se han declarado en esa función, es decir una copia de la tabla de simbolos de manera que cuando se esté ejecutando la función, esas variables y todo está en esa pila. Si esa función se llama a si misma, mas abajo se crea otro registro con otra copia, no se repite el código porque es el mismo, pero si se crea otra copia de los datos. En la medida que se van llamando fuciones, es pila se va ampliando y va crecienod hacia abajo. Ciuando termina de ejecutarse, se libera el registro de activación , la pila va creciendo y se vuelve hacia arriba, pues al final del programa la pila esa debe estar vacía. Un registro de activación se asociada a la LLAMADA DE LA FUNCIÓN.

Si hay una variable local que no esta en su registro de activación se busca, ¿por donde?, pues por donde sea. Al final del todo, pone los datos dinámicos del programa, las estructuras dinámicas, que son los datos que hemos creado nosotros y que van creciendo y decreciendo las pone al final del mapa y van a crecer hacia arriba.

Después, el cargador haría otra cosa………//buscarlo no me ha dado tiempo a escribirlo…

Hay 3 formas de hacer un ejecutable: con el cargador que tiene una carga dinámica y estatica y a través del compilador.

En la encuadernación hay 2 referencias, una que es estatica, que no esta en ejecución, y lo que hace el encuardernador es buscar en la biblioteca, coge lo que corresponda, lo trae y lo empaqueta. Cuando se genera el ejecutable, ahí está todo. La otra es la dinámica, que sabe donde está, no lo pone, directamente pone una llamada á, y cuando se está ejecutando el programa, se llama, lo coge y lo pone. Diferencias, tanto la que resuelve el compilado como la estática, al final mi código ocupa mas espacio porque lo tiene todo, mientras que la dinámica no lo mete hasta que no lo ejecute y por tante es mas chico. Eso es una ventaja pero la desventaja es que en ejecución tarda más tiempo porque tiene que hacer la carga en ejecución y la