



TECNOLÓGICO  
NACIONAL DE MÉXICO®



## Apuntes de la asignatura

Integrantes:

- Guerrero Verde Geovanni 161080010
- Rojas Hernández Axel Joel 181080176
- Sánchez Morales Luis Daniel 161080019
- Oscar Lucero Zavaleta 171080170

Materia: Lenguajes y Automatas II (Compiladores)  
Grupo: ISC-7AV SCD1016



## ~APUNTES DE LOS CONCEPTOS~

### **Compilador**

Un compilador es de manera simple un programa que puede leer un programa en un lenguaje (Lenguaje "Fuente") y traducirlo en un programa equivalente en otro lenguaje (Lenguaje "Destino").

### **Intérprete**

Este a diferencia del compilador no crea un programa destino, sino que da la ilusión de ejecutar directamente el código.

El programa destino en lenguaje máquina que produce un compilador es, por lo general, más rápido que un intérprete al momento de asignar las entradas a las salidas. No obstante, por lo regular, el intérprete puede ofrecer mejores diagnósticos de error que un compilador, ya que ejecuta el programa fuente instrucción por instrucción.

### **Preprocesador**

El preprocesador es un programa que es invocado por el compilador antes de que se comience la traducción del código, este puede eliminar código innecesario como los comentarios, incluir archivos y ejecutar sustituciones de macros.

### **Macro**

Son porciones de código que el compilador pega en el lugar de invocación, también sirve para hacer más sencillas tareas complicadas o simplemente economizar código.



TECNOLÓGICO  
NACIONAL DE MÉXICO®



posicion = inicial + velocidad \* 60

Analizador léxico

$\langle \text{id}, 1 \rangle \langle = \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle * \rangle \langle 60 \rangle$

Analizador sintáctico

$\langle \text{id}, 1 \rangle = + * 60$   
 $\langle \text{id}, 2 \rangle \langle \text{id}, 3 \rangle$

Analizador semántico

$\langle \text{id}, 1 \rangle = + * \text{inttofloat}$   
 $\langle \text{id}, 2 \rangle \langle \text{id}, 3 \rangle$   
60

Generador de código intermedio

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Optimizador de código

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Generador de código

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

## Analizador Léxico

Su principal función consiste en leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis.

## Análisis Sintactico



El parser (analizador sintáctico) utiliza los primeros componentes de los tokens producidos por el analizador de léxico para crear una representación intermedia en forma de árbol que describa la estructura gramatical del flujo de tokens.

### **Análisis Semántico**

El analizador semántico utiliza el árbol sintáctico y la información en la tabla de símbolos para comprobar la consistencia semántica del programa fuente con la definición del lenguaje. También recopila información sobre el tipo y la guarda, ya sea en el árbol sintáctico o en la tabla de símbolos, para usarla más tarde durante la generación de código intermedio.

### **Generación de código intermedio**

Después del análisis sintáctico y semántico del programa fuente, muchos compiladores generan un nivel bajo explícito, o una representación intermedia similar al código máquina, que podemos considerar como un programa para una máquina abstracta.

### **Optimizador de código**

La optimización de código es el conjunto de fases de un compilador que transforma un fragmento de código en otro fragmento con un comportamiento equivalente y que se ejecuta de forma más eficiente, es decir, usando menos recursos de cálculo como memoria o tiempo de ejecución.

### **Generador de código.**

El generador de código recibe como entrada una representación intermedia del programa fuente y la asigna al lenguaje destino. Si el lenguaje destino es código máquina, se seleccionan registros o ubicaciones (localidades) de memoria para cada una de las variables que utiliza el programa.



TECNOLÓGICO  
NACIONAL DE MÉXICO®

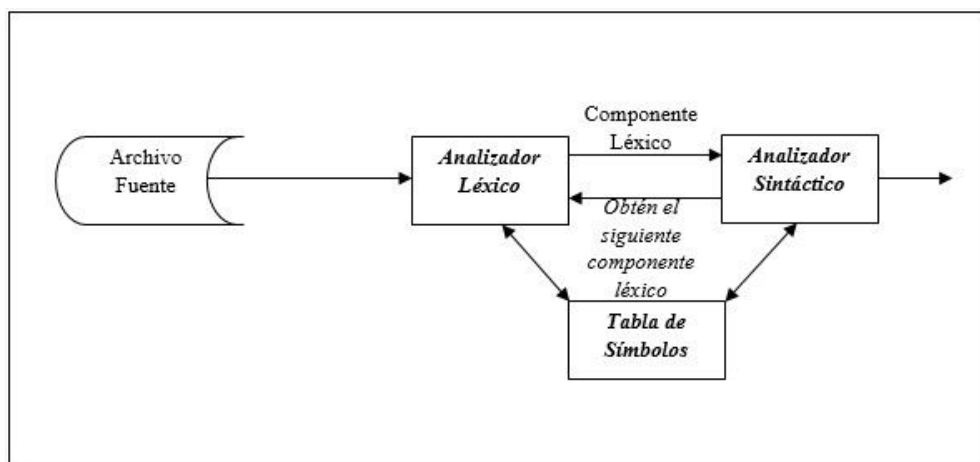


Los compiladores son programas de computadora que traducen de un lenguaje a otro. Un compilador toma como su entrada un programa escrito en lenguaje fuente y produce un programa equivalente escrito en lenguaje objeto.

También sabemos por el semestre pasado es que un compilador tiene fases al momento de ejecutar algo, es útil pensar que en estas fases como piezas separadas del compilador, y pueden en realidad escribirse como operaciones como operaciones especificadas y las integran:

- Análisis Léxico
- Análisis Sintáctico
- Análisis Semántico
- Generación y Optimización de código intermedio
- Generación de código objeto

Su principal función consiste en leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis. En la figura 1 se puede apreciar el esquema de una interacción que se aplica convirtiendo el analizador léxico en una subrutina o corrutina del analizador sintáctico. Recibida la orden "obtener el siguiente componente léxico" del analizador sintáctico, el analizador léxico lee los caracteres de entrada hasta que pueda identificar el siguiente componente léxico.



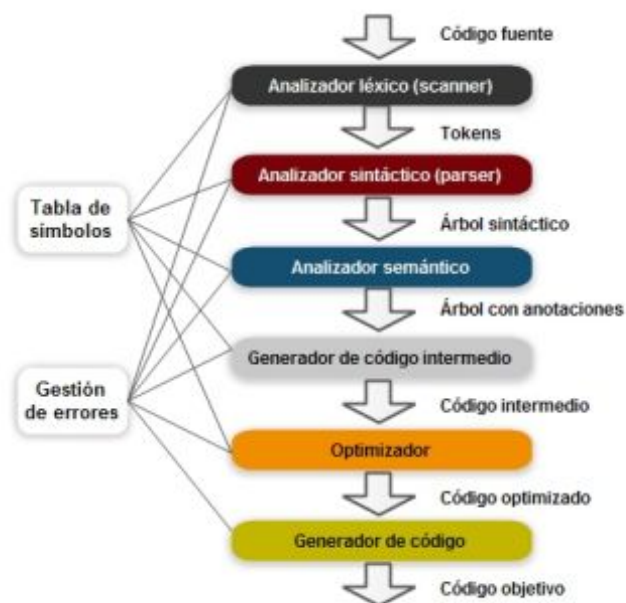
## Semana 5

-El análisis léxico En este tema vamos a proceder a definir y comprender todas las tareas que realiza el analizador léxico y que son clave para el correcto funcionamiento del compilador. Como vemos en la figura, tiene

como entrada el código fuente del lenguaje de programación que acepta el compilador y como salida, proporciona al analizador sintáctico los tokens.

¿Qué es un token?

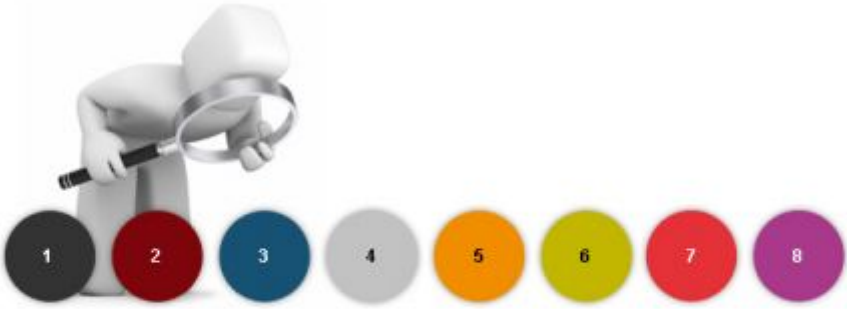
Es una agrupación de caracteres reconocidos por el analizador léxico que constituyen los símbolos con los que se forman las sentencias del lenguaje. Lo que el analizador léxico devuelve al analizador sintáctico es el nombre de ese símbolo junto con el valor del atributo (si ese token lo necesita, ya que no todos los tokens llevan atributo, como por ejemplo: una palabra reservada "if")




Funciones del analizador léxico y sus ventajas El analizador léxico realiza varias funciones, siendo la fundamental la de agrupar los caracteres que va leyendo uno a uno del programa fuente y formar los tokens. Las otras funciones que realiza el analizador léxico son:




TECNOLÓGICO  
NACIONAL DE MÉXICO®






1


Gestionar el fichero que contiene el código fuente; entendiéndose por ello el abrirlo, leerlo y cerrarlo.






2

Eliminar comentarios, tabuladores, espacios en blanco, saltos de línea.







TECNOLÓGICO  
NACIONAL DE MÉXICO®



Ventajas de contar con un analizador léxico:

1ª ventaja: Se simplifica el diseño, puesto que hay una herramienta especializada en el tratamiento del fichero que contiene el código fuente.

2ª ventaja Aumenta la portabilidad del compilador, pudiendo tener versiones diferentes para distintos formatos del texto de código fuente (ASCII, EBCDIC, etc.).

3ª ventaja Mejora la eficiencia al ser una herramienta especializada en el tratamiento de caracteres.

4ª ventaja Detección de determinados errores fáciles de corregir a este nivel (5.25 por 5,25).

-Un analizador sintáctico o parser (viene del inglés: parse - analizar una cadena o texto en componentes sintácticos lógicos) es un programa que normalmente es parte de un compilador. El compilador se asegura de que el código se traduce correctamente a un lenguaje ejecutable. La tarea del analizador es, en este caso, la descomposición y transformación de las entradas en un formato utilizable para su posterior procesamiento. Se analiza una cadena de instrucciones en un lenguaje de programación y luego se descompone en sus componentes individuales.

**¿Cómo funciona?**



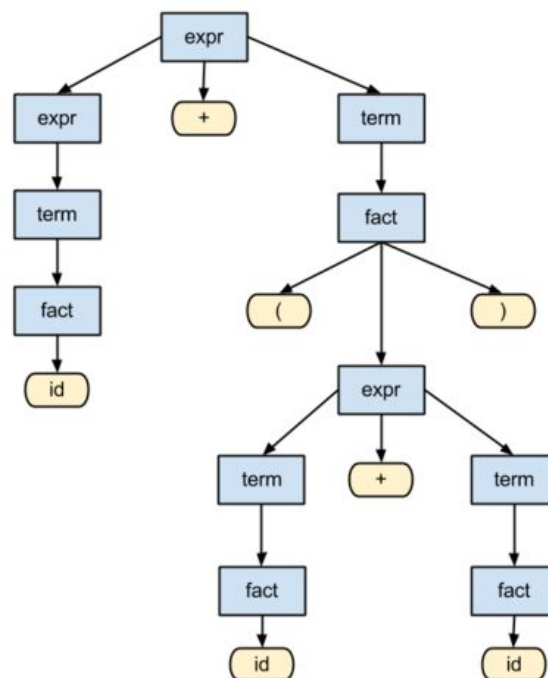


TECNOLÓGICO  
NACIONAL DE MÉXICO®



Para analizar un texto, los analizadores suelen utilizar un analizador léxico separado (llamado lexer), que descompone los datos de entrada en fichas (símbolos de entrada como palabras). Los Lexers son por lo general máquinas de finitas, que siguen la gramática regular y por lo tanto aseguran un desglose adecuado. Los tokens obtenidos de esta manera sirven como caracteres de entrada para el analizador sintáctico.

El analizador actual maneja la gramática de los datos de entrada, realiza un análisis sintáctico de éstos y como regla general crea un árbol de sintaxis (árbol de análisis). Esto se puede utilizar para el procesamiento posterior de los datos, por ejemplo, la generación de código por un compilador o ejecutado por un intérprete (traductor). Por lo tanto, el analizador es el software que comprueba, procesa y envía las instrucciones del código fuente.



-SEMANA 6

## Tipos de analizadores



Hay básicamente dos métodos de análisis diferentes, análisis de arriba hacia abajo (top-down) y análisis de abajo hacia arriba (bottom-up). Éstos difieren generalmente en el orden en el que se crean los elementos del árbol sintáctico.

-De arriba a abajo: En el método top-down, el analizador trabaja en un método orientado a objetivos, lo que significa que busca a partir del símbolo de inicio de la sintaxis y busca una derivación sintáctica adecuada. Por lo tanto, el árbol de análisis se desarrolla de arriba hacia abajo en la dirección de un desglose cada vez más detallado.

-De abajo hacia arriba: El analizador ascendente comienza con el símbolo de la cadena de entrada e intenta establecer relaciones sintácticas cada vez mayores. Esto se hace hasta que el símbolo de inicio de la gramática se ha alcanzado.

## Aplicaciones

Un analizador sintáctico se utiliza a menudo para convertir texto en una nueva estructura, por ejemplo, un árbol sintáctico, que expresa la disposición jerárquica de los elementos. En las siguientes aplicaciones el uso de un analizador es usualmente esencial:

-La lectura de un lenguaje de programación es realizada por un analizador. Proporciona una estructura de datos al compilador, con la que se puede generar el código máquina o bytecode.

-El código HTML es al principio sólo una cadena de caracteres para un ordenador que debe ser analizada por el analizador contenido en el navegador web. Proporciona una descripción de la página web como una



estructura de datos que puede ser proyectada por un motor de diseño en la pantalla.

- Los analizadores especiales de XML son responsables del análisis de los documentos XML y preparan la información contenida en ellos para su uso posterior.

- Los analizadores de URI descomponen esquemas complejos tales como URLs en su estructura jerárquica.

- Los motores de búsqueda como Google extraen (analizan) texto relevante para ellos de las páginas web descargadas con rastreadores. Se procesan y los datos analizados se pueden utilizar para la navegación.

## Semana 7

En informática , un analizador LALR [a] o un analizador LR Look-Ahead es una versión simplificada de un analizador LR canónico , para analizar (separar y analizar) un texto de acuerdo con un conjunto de reglas de producción especificadas por una gramática formal para una computadora idioma . ("LR" significa derivación de izquierda a derecha, más a la derecha ).

El analizador sintáctico LALR fue inventado por Frank DeRemer en su tesis doctoral de 1969, Traductores prácticos para lenguajes LR (k) , [1] en su tratamiento de las dificultades prácticas en ese momento de implementar analizadores sintácticos LR (1). Demostró que el analizador LALR tiene más poder de reconocimiento de lenguaje que el analizador LR (0), mientras que requiere el mismo número de estados que el analizador LR (0) para un idioma que pueda ser reconocido por ambos analizadores. Esto hace que el analizador LALR sea una alternativa eficiente en memoria al analizador LR (1) para lenguajes que son LALR. También se comprobó que existen lenguajes LR (1) que no son LALR. A pesar de esta debilidad, el poder del analizador



LALR es suficiente para muchos lenguajes informáticos convencionales, [2] incluido Java, [3] aunque las gramáticas de referencia para muchos idiomas no son LALR debido a que son ambiguas . [2]

La disertación original no proporcionó ningún algoritmo para construir tal analizador dada una gramática formal. Los primeros algoritmos para la generación de analizadores sintácticos LALR se publicaron en 1973. [4] En 1982, DeRemer y Tom Pennello publicaron un algoritmo que generaba analizadores LALR altamente eficientes en memoria. [5] Los analizadores sintácticos LALR se pueden generar automáticamente a partir de una gramática mediante un generador de analizadores sintácticos LALR como Yacc o GNU Bison . El código generado automáticamente puede aumentarse mediante código escrito a mano para aumentar la potencia del analizador resultante.

### **Contenido**

En 1965, Donald Knuth inventó el analizador sintáctico LR ( L EFT a derecha, R ightmost ). El analizador LR puede reconocer cualquier lenguaje determinista libre de contexto en tiempo lineal delimitado. [6] La derivación más a la derecha tiene requisitos de memoria muy grandes y la implementación de un analizador LR no era práctica debido a la memoria limitada de las computadoras en ese momento. Para abordar esta deficiencia, en 1969, Frank DeRemer propuso dos versiones simplificadas del analizador LR, a saber, Look-Ahead LR (LALR) [1] y el analizador Simple LR que tenía requisitos de memoria mucho más bajos a costa de menos capacidad de reconocimiento de idioma, siendo el analizador LALR la alternativa más poderosa. [1] En 1977, se inventaron optimizaciones de memoria para el analizador LR [7], pero aún así, el analizador LR era menos eficiente en memoria que las alternativas simplificadas.



En 1979, Frank DeRemer y Tom Pennello anunciaron una serie de optimizaciones para el analizador LALR que mejorarían aún más la eficiencia de su memoria. [8] Su trabajo fue publicado en 1982. [5]

Generalmente, el analizador LALR se refiere al analizador LALR (1), [b] al igual que el analizador LR generalmente se refiere al analizador LR (1). El "(1)" denota una anticipación de un token, para resolver las diferencias entre los patrones de reglas durante el análisis. De manera similar, hay un analizador LALR (2) con búsqueda anticipada de dos tokens y analizadores LALR (k) con búsqueda de token k, pero estos son raros en el uso real. El analizador LALR se basa en el analizador LR (0), por lo que también se puede denotar  $LALR(1) = LA(1)LR(0)$  (1 token de búsqueda anticipada, LR (0)) o más generalmente  $LALR(k) = LA(k)LR(0)$  (k tokens de anticipación, LR (0)). De hecho, existe una familia de dos parámetros de analizadores sintácticos  $LA(k)LR(j)$  para todas las combinaciones de j y k, que pueden derivarse de la  $LR(j+k)$  del analizador, [9], pero estos no ven el uso práctico.

Al igual que con otros tipos de analizadores LR, un analizador LALR es bastante eficiente para encontrar el único análisis ascendente correcto en un solo escaneo de izquierda a derecha sobre el flujo de entrada, porque no necesita usar retroceso. Al ser un analizador de búsqueda anticipada por definición, siempre utiliza una búsqueda anticipada, siendo LALR (1) el caso más común.

## **Analizadores LR**



El analizador LALR (1) es menos potente que el analizador LR (1) y más potente que el analizador SLR (1), aunque todos usan las mismas reglas de producción . La simplificación que introduce el analizador LALR consiste en fusionar reglas que tienen conjuntos de elementos del kernel idénticos , porque durante el proceso de construcción del estado LR (0) no se conocen las búsquedas anticipadas. Esto reduce el poder del analizador porque no conocer los símbolos de anticipación puede confundir al analizador en cuanto a qué regla gramatical elegir a continuación, lo que resulta en reducir / reducir conflictos . Todos los conflictos que surgen al aplicar un analizador LALR (1) a una gramática LR (1) inequívoca son conflictos de reducción / reducción. El analizador SLR (1) realiza una fusión adicional, lo que introduce conflictos adicionales.

---

El ejemplo estándar de una gramática LR (1) que no se puede analizar con el analizador LALR (1), que presenta un conflicto de reducción / reducción de este tipo, es: [10] [11]

$$\begin{aligned} S &\rightarrow a E c \\ &\rightarrow a F d \\ &\rightarrow b F c \\ &\rightarrow b E d \\ E &\rightarrow e \\ F &\rightarrow e \end{aligned}$$

En la construcción de la tabla LALR, dos estados se fusionarán en un estado y luego se encontrará que las búsquedas anticipadas son ambiguas. El único estado con lookaheads es:

$$E \rightarrow e. \{\text{discos compactos}\}$$



$F \rightarrow e. \{\text{discos compactos}\}$

Un analizador LR (1) creará dos estados diferentes (con búsquedas anticipadas no conflictivas), ninguno de los cuales es ambiguo. En un analizador sintáctico LALR, este estado tiene acciones en conflicto (dada la anticipación  $c$  o  $d$ , reducir a  $E$  o  $F$ ), un "reducir / reducir el conflicto"; la gramática anterior será declarada ambigua por un generador de analizador LALR y se informarán los conflictos.

Para recuperarse, esta ambigüedad se resuelve eligiendo  $E$ , porque aparece antes que  $F$  en la gramática. Sin embargo, el analizador resultante no podrá reconocer la secuencia de entrada válida  $b e c$ , ya que la secuencia ambigua  $e c s e$  reduce a  $(E \rightarrow e) c$  la correcta  $(F \rightarrow e) c$ , pero  $b E c$  no está en la gramática.

### Analizadores de LL [ [editar](#) ]

Los analizadores sintácticos LALR (  $j$  ) son incomparables con los analizadores sintácticos LL (  $k$  ) : para cualquier  $j$  y  $k$  ambos mayores que 0, existen gramáticas LALR (  $j$  ) que no son gramáticas LL (  $k$  ) y viceversa. De hecho, es indecidible si una gramática LL (1) dada es LALR (  $k$  ) para cualquier

$\{\backslash \text{Displaystyle } k > 0\}$

Dependiendo de la presencia de derivaciones vacías, una gramática LL (1) puede ser igual a una gramática SLR (1) o LALR (1). Si la gramática LL (1) no tiene derivaciones vacías, es SLR (1) y si todos los símbolos con derivaciones vacías tienen derivaciones no vacías, es LALR (1). Si existen símbolos que solo tienen una derivación vacía, la gramática puede o no ser LALR (1).



El concepto clave de LLVM es una representación "intermedia" (IR) de bajo nivel de su programa. Este IR está aproximadamente al nivel de código de ensamblador, pero contiene más información para facilitar la optimización.

La potencia de LLVM proviene de su capacidad de diferir la compilación de esta representación intermedia en una máquina de destino específica hasta justo antes de que se ejecute el código. Se puede utilizar un enfoque de compilación justo a tiempo (JIT) para que una aplicación produzca el código que necesita justo antes de que lo necesite.

En muchos casos, tiene más información en el momento en que se ejecuta el programa que en la oficina central, por lo que el programa puede ser mucho más optimizado.

Para comenzar, puede compilar un programa C++ en una única representación intermedia, luego compilarlo en múltiples plataformas desde ese IR.

También puede probar la demostración de Kaleidoscope, que le guiará en la creación de un nuevo idioma sin tener que escribir un compilador, simplemente escriba el IR.

En aplicaciones de rendimiento crítico, la aplicación puede escribir esencialmente su propio código que necesita para ejecutarse, justo antes de que necesite ejecutarlo.

## **SEMANA 10 (ANTEPROYECTO)**

### **Como usa LLVM**

Siempre se recomienda ir al sitio web oficial de LLVM y seguir las guías de instalación según su sistema operativo.

Si está trabajando en posix, en resumen, debe agregar uno de los repositorios oficiales de paquetes LLVM . Por ejemplo, si trabaja en Ubuntu





Xenial (16.04) agrega una entrada deb y deb-src a su archivo /etc/apt/sources.list :

```
$ sudo su
```

```
$ echo deb http://apt.lvm.org/xenial/ llvm-toolchain-xenial-4.0 main \ >>  
/etc/apt/sources.list
```

```
$ echo deb-src http://apt.lvm.org/xenial/ llvm-toolchain-xenial-4.0 main \ >>  
/etc/apt/sources.list
```

Y una vez que lo haces, la instalación es tan simple como llamar.

```
$ sudo apt update
```

```
$ sudo apt install clang-X
```

donde X es la versión que está buscando (4.0 está vigente al momento de escribir esta publicación).

Tenga en cuenta que clang es un compilador de C / C ++ escrito sobre LLVM (y en realidad ahora está alojado automáticamente) y viene junto con todas las bibliotecas de LLVM. Una vez que haces eso puedes ir a cualquier tutorial y comenzar a codificar.

Si lo desea puede instalar bibliotecas LLVM manualmente. Para eso, solo tiene que apt install llvm-Y donde Y es la biblioteca que está buscando. Sin embargo, sí recomiendo compilar LLVM usando proyectos con clang.

Una vez que hagas eso deberías tener la herramienta llvm-config . Es muy útil obtener los indicadores del compilador necesarios para la compilación correcta del proyecto LLVM. Así que la primera prueba que funcionó sería llamando



```
$ llvm-config-4.0 --cxxflags --libs engine
```

```
-I/usr/lib/llvm-4.0/include -std=c++0x -gsplit-dwarf -Wl,-fuse-ld=gold -fPIC  
-fvisibility-inlines-hidden -Wall -W -Wno-unused-parameter -Wwrite-strings  
-Wcast-qual -Wno-missing-field-initializers -pedantic -Wno-long-long  
-Wno-maybe-uninitialized -Wdelete-non-virtual-dtor -Wno-comment  
-Werror=date-time -std=c++11 -ffunction-sections -fdata-sections -O2 -g  
-DNDEBUG -fno-exceptions -D_GNU_SOURCE -D__STDC_CONSTANT_MACROS  
-D__STDC_FORMAT_MACROS -D__STDC_LIMIT_MACROS
```

```
-ILLVM-4.0
```

Puede obtener un conjunto diferente de banderas, no se preocupe por ello. Siempre y cuando no falle con el command not found , debería estar bien.

El siguiente paso es probar la biblioteca LLVM real en sí. Así que vamos a crear un simple archivo `llvmtest.cpp` :

```
#include <iostream>
```

```
#include "llvm/IR/LLVMContext.h"
```

```
int main() {
```

```
    llvm::LLVMContext context;
```

```
    std::cout << &context << std::endl;
```

```
    return 0;
```

```
};
```



Tenga en cuenta que uso `std::cout` para que utilicemos la variable de context (para que el compilador no la elimine durante la fase de compilación). Ahora compila el archivo con

```
$ clang++-4.0 -o llvmtest `llvm-config-4.0 --cxxflags --libs engine` llvmtest.cpp
```

y probarlo

```
$ ./llvmtest
```

```
0x7ffd85500970
```



## CONCLUSIÓN (Oscar).

### Introducción a compiladores.

Los principios y técnicas que se usan en la escritura de compiladores se pueden emplear en muchas otras áreas. Se basan en los conceptos de teoría de autómatas y lenguajes formales que se están exponiendo en la parte teórica, y constituyen un campo de aplicación práctica bastante directo.

El objetivo es, básicamente es traducir un programa (o texto) escrito en un lenguaje “fuente”, que llamaremos programa fuente, en un equivalente en otro lenguaje denominado “objeto”, al que llamaremos programa o código objeto. Si el programa fuente es correcto (pertenece al lenguaje formado por los programas correctos), podrá hacerse tal traducción; si no, se deseara obtener un mensaje de error (ovarios) que permita determinar lo más claramente posible los orígenes de la incorrección. La traducción podrá hacerse en dos formas:

**interpretación:** la traducción se hace “frase a frase”

**compilación:** la traducción se hace del texto completo

Concentramos nuestra atención en el segundo de los métodos

Mientras tanto “LLVM” es un proyecto de código abierto que busca la creación de compiladores para cualquier lenguaje de programación, proporcionando la infraestructura necesaria para su desarrollo.

## Análisis léxico

La cadena de entrada se recibe como una sucesión de caracteres. El análisis léxico agrupa los caracteres en secuencias con significado colectivo y mínimo en el lenguaje, llamadas componentes léxicos (palabras o “token”), conciertos atributos léxicos. En el ejemplo, se detectarían 7:



1.el identificador posición

2.el operador de asignación: =

3.el identificador inicial

4.el operador +

5.el identificador velocidad

6.el operador \*

7.la constante numérica 60

Cada vez que se detecta un nuevo identificador, se anota una entrada en la tabla de símbolos. El lenguaje de entrada tendrá un catálogo de componentes posibles, qué se podrán codificar (por ejemplo, mediante constantes enteras).