******

***Universidad Tecnológica Nacional - Facultad Regional Buenos Aires***

**Sintaxis y Semantica de los Lenguajes**

**2018**

**Tema: Compilador Lenguaje Micro**

**Curso:** K2003

**Profesor:** Oscar Bruno

|  |  |
| --- | --- |
| **Alumno** | **Legajo** |
| Romano, Leandro | 164.294-7 |
| Sarrelli, Julian | 163.969-9 |
| Korbenfeld, Florencia | 159.579-9 |
| Gandolfo, Lazaro | 163.291-7 |
|  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Fecha** | **Nota** | **Observaciones** | **Fecha Devol.** | **Firma Docente** |
| **Entrega** |  |  |  |  |  |
| **Corrección** |  |  |  |  |  |

**Compilador**

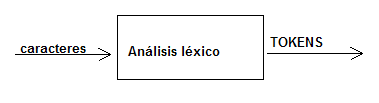
Es aquel traductor que tiene como entrada una sentencia en lenguaje formal y como salida tiene un fichero ejecutable, es decir, realiza una traducción de un código de alto nivel a código máquina (también se entiende por compilador aquel programa que proporciona un fichero objeto en lugar del ejecutable final).

EL ANÁLISIS DEL PROGRAMA FUENTE

En la compilación, el análisis está formado por tres fases:

1. el Análisis Léxico: El Análisis Léxico detecta los diferentes elementos básicos que constituyen un programa fuente, como: identificadores, palabras reservadas, constantes, operadores y caracteres de puntuación (lexemas) .

El Análisis Léxico solo se ocupa de los Lenguajes Regulares (TOKENS) que forman parte del Lenguaje de Programación.



El analizador léxico se lo denomina scanner.

1. el Análisis Sintáctico: El Análisis Sintáctico trabaja con los tokens detectados durante el Análisis Léxico. Tendrá la capacidad de determinar si las construcciones que componen el programa son sintácticamente correctas pero, no podrá determinar si el programa, en su totalidad, es sintácticamente correcto.

El analizador sintáctico se lo denomina *parser*.

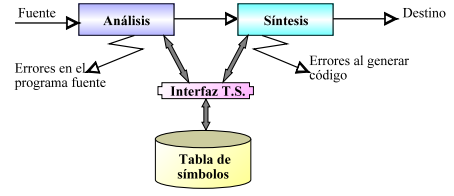
1. el Análisis Semántico. El Análisis Semántico realiza diferentes tareas, completando lo que hizo el Análisis Sintáctico. Una de estas importantes tareas es la “verificación de tipos”, para que cada operador trabaje sobre operandos permitidos según la especificación del Lenguaje de Programación.

Esto se hace con las rutinas semánticas que tienen como tarea dos funciones:

1)Chequean la semántica estática de cada construcción; es decir, verifican que la construcción analizada sea legal y que tenga un significado. Verifican que las variables involucradas estén definidas, que los tipos sean correctos, etc.

2) Si la construcción es semánticamente correcta, las rutinas semánticas también hacen la traducción; es decir, generan el código para una Máquina Virtual que, a través de sus instrucciones, implementa correctamente la construcción analizada.

Luego del análisis viene la SÍNTESIS, que, a partir de esta representación intermedia, construye el programa objeto.



**Existen dos formas principales de implementar un Scanner:**

a) A través de la utilización de un programa auxiliar tipo flex/lex, en el que los datos son tokens representados mediante Expresiones Regulares

b) Mediante la construcción de una rutina basada en el diseño de un AFD.

**Existen dos formas fundamentales de Análisis Sintáctico:**

1. Análisis Sintáctico Descendente (conocido como top-down), que permite ser construido por un programador,
2. Análisis Sintáctico Ascendente (conocido como bottom-up), que requiere la ayuda de un programa especializado tipo yacc/bison.

**Compilador para Lenguaje Micro utilizando Automata**



*Nosotros elegiremos para la realización del compilador los programas auxiliares lex y yacc.*

**Resumen de flex/lex:**

Flex es un una herramienta que permite generar analizadores léxicos. A partir de un conjunto de expresiones regulares, Flex busca concordancias en un fichero de entrada y ejecuta acciones asociadas a estas expresiones.

Los ficheros de entrada de Flex (normalmente con la extensión .l) siguen el siguiente esquema:

definiciones

%%

reglas

%%

código de usuario (código que se deja tal cual escribe el programador. Es opcional)

La definición esta compuesto por:

**Nombre expresión regular**

Ejemplo

digito [0-9] *(véase que [0-9] es la definición de todos los números del 0 al 9)*

Las reglas están compuestas por:

**patrón1** {acción1}

**patrón2**  {acción2}

donde:

**patrón**: expresión regular

**acción**: código C con las acciones a ejecutar cuando se encuentre concordancia del patrón con el texto de entrada.

Flex genera como salida un fichero fuente en C, ‘lex.yy.c’, que define una función ‘yylex()’. Este fichero se compila y se enlaza con la librería de Flex para producir un ejecutable. Cuando se arranca el fichero ejecutable, este analiza su entrada en busca de casos de las expresiones regulares. Siempre que encuentra uno, ejecuta el código C correspondiente.

*Condiciones de arranque (sensibilidad al contexto) :Flex dispone de un mecanismo para activar reglas condicionalmente. Cualquier regla cuyo patrón se prefije con "" únicamente estará activa cuando el analizador se encuentre en la condición de arranque llamada "sc".*

*Acotación:*

*Flex requiere un formato bastante estricto de su fichero de entrada. En particular los caracteres no visibles (espacios en blanco, tabuladores, saltos de línea) fuera de sitio 8 Introducción a Flex y Bison causan errores difíciles de encontrar. Sobre todo es muy importante no dejar líneas en blanco de más ni empezar reglas con espacios en blanco o tabuladores.*

**Resumen de BISON/yacc**

Bison es un generador de analizadores sintácticos de propósito general que convierte una descripción para una gramática independiente del contexto (en realidad de una subclase de éstas, las LALR) en un programa en C que analiza esa gramática.

Un fuente de Bison (normalmente un fichero con extensión .y) describe una gramática. El ejecutable (.c) que se genera indica si un fichero de entrada dado pertenece o no al lenguaje generado por esa gramática. La forma general de una gramática de Bison es la siguiente:

**%{ declaraciones en C %}**

**Declaraciones de Bison**

**%%**

**Reglas gramaticales**

**%%**

**Código C adicional**

Los ‘%%’, ‘%{‘ y ‘%}’ son signos de puntuación que aparecen en todo archivo de gramática de Bison para separar las secciones.

Las declaraciones en C pueden definir tipos y variables utilizadas en las acciones. Puede también usar comandos del preprocesador para definir macros que se utilicen ahí, y utilizar #include para incluir archivos de cabecera que realicen cualquiera de estas cosas.

Los símbolos terminales de la gramática se denominan en Bison tokens y deben declararse en la sección de definiciones. Por convención se suelen escribir los tokens en mayúsculas y los símbolos no terminales en minúsculas.

Una regla gramatical de Bison tiene la siguiente forma general:

resultado: componentes... ;

donde resultado es el símbolo no terminal que describe esta regla y componentes son los diversos símbolos terminales y no terminales que están reunidos por esta regla. Por ejemplo,

exp: exp ‘+’ exp

;

El fuente de Bison se convierte en una función en C llamada yyparse. Aquí describimos las convenciones de interfaz de yyparse y las otras funciones que éste necesita usa.

**Yyparse():** Se llama a la función yyparse para hacer que el análisis comience. Esta función lee tokens, ejecuta acciones, y por último retorna cuando se encuentre con el final del fichero o un error de sintaxis del que no puede recuperarse. Usted puede también escribir acciones que ordenen a yyparse retornar inmediatamente sin leer más allá. El valor devuelto por yyparse es 0 si el análisis tuvo éxito (el retorno se debe al final del fichero). El valor es 1 si el análisis falló (el retorno es debido a un error de sintaxis)

**Compilador de Lenguaje Micro utilizando Lex y Yacc**

* **LEX**

****

* **YACC**

****

****

****

**Lenguaje Micro:**

**Definición informal:**

- El único tipo de dato es entero.

- Todos los identificadores son declarados implícitamente y con una longitud máxima de 32 caracteres.

- Los identificadores deben comenzar con una letra y están compuestos de letras y dígitos.

- Las constantes son secuencias de dígitos (números enteros).

- Hay dos tipos de sentencias:

Asignación

**ID := Expresión;**

Expresión es infija y se construye con identificadores, constantes y los operadores + y –; los paréntesis están permitidos.

Entrada/Salida

**leer (lista de IDs);**

**escribir (lista de Expresiones);**

- Cada sentencia termina con un "punto y coma" (;). El cuerpo de un programa está delimitado por inicio y fin.

- inicio, fin, leer y escribir son palabras reservadas y deben escribirse en minúscula.

TOKENS

