UNIVERSIDAD NACIONAL AUTÓNOMA DE NICARAGUA

UNAN-LEÓN

FACULTAD DE CIENCIAS Y TECNOLOGÍAS

DEPARTAMENTO DE COMPUTACIÓN

INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Tesis para optar al Título de Ingeniero en Sistemas de Información

PROYECTO DE APOYO PARA LA ASIGNATURA "DISEÑO DE COMPILADORES"

Autor: Br. Francisco Ramón Vargas Rodríguez

León, Febrero de 2011

"A LA LIBERTAD POR LA UNIVERSIDAD"

UNIVERSIDAD NACIONAL AUTÓNOMA DE NICARAGUA

UNAN-LEÓN

FACULTAD DE CIENCIAS Y TECNOLOGÍAS

DEPARTAMENTO DE COMPUTACIÓN

INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Tesis para optar al Título de Ingeniero en Sistemas de Información

PROYECTO DE APOYO PARA LA ASIGNATURA "DISEÑO DE COMPILADORES"

Autor: **Br. Francisco Ramón Vargas Rodríguez**

Tutor: **MSc. Ricardo Espinoza**

Colaborador: **MSc. William Noel Martínez**

León, Febrero de 2011

"A LA LIBERTAD POR LA UNIVERSIDAD"

AGRADECIMIENTO

Le agradezco primeramente a DIOS por brindarme la vida, salud, entendimiento, fortaleza y sabiduría, gracias DIOS Padre Celestial, te alabo y te bendigo, por estar conmigo en cada paso que doy, por fortalecer mi corazón e iluminar mi mente y por haber puesto en mi camino a aquellas personas que han sido mi soporte y compañía durante todo el periodo de mis estudios para desarrollar y culminar este trabajo investigativo y cumplir las metas propuestas en mi vida.

A mi familia principalmente a mi tía Odily que en paz descanse, a mama Lila, por sus enseñanzas, DIOS la tenga en su santa gloria, a mis padres y hermanos, a mis primas Isabel y Arlen y a mis demás tíos y primos que me han ayudado en todo lo que han tenido a su alcance para ayudarme a salir adelante y culminar mis estudios universitarios.

De igual manera mi más sincero agradecimiento a mi tutor y Director del Departamento de Computación prof. MSc. Ricardo Espinoza, un agradecimiento especial al prof. MSc. William Noel Martínez, por la colaboración, paciencia, apoyo brindados desde siempre y sobre todo por esa gran amistad que me brindó y me brinda, por escucharme y aconsejarme siempre, gracias profe, por brindarme su ayuda cuando más la necesitaba, por ser una persona con la que puedo contar siempre, por el cariño que me brinda y los ánimos que me da, por los momentos en los que más que un profesor se comportó como un amigo.

A mi novia Ana Sugey Galeano Bautista y a su mamá Doña Ana Celia Bautista. A Sugey, por ser la persona que ha compartido el mayor tiempo a mi lado, porque en su compañía las cosas malas se convierten en buenas, la tristeza se transforma en alegría y la soledad no existe. Y a doña Celia, por ser como una madre para mí, a las dos les doy gracias por brindarme su amor, comprensión y ánimo para llegar a realizar mi tesis.

A mi amiga Ivonne Maria Martínez Montenegro y a mi amigo Rudy Francisco Mairena, a quienes aprecio y agradezco mucho, por ser tan buenas personas conmigo, brindándome siempre su apoyo, ánimo y colaboración en todo momento y sobre todo cuando más necesitaba de ellos, sin poner nunca peros o darme negativas, sino todo lo contrario, me han brindado siempre una sonrisa.

Al Sr. César Díaz, a don Sergio Amaya, y a todas las personas que de manera directa o indirecta me ayudaron a la realización de mi trabajo investigativo brindándome su apoyo incondicional.

DEDICATORIA

Mi sacrificio, empeño, amor, fe, esperanza y confianza infinita a mi Padre Celestial DIOS todo poderoso, a mi madre Santisima la Virgen Maria y a mi Señor Jesucristo quien es mi amigo fiel y la única persona que jamás me ha dejado solo por muy difícil que sean los tiempos.

A la memoria de mi mama Lila y mi tía Odily, DIOS las tenga en su santa gloria.

A mi hermano Pedro Antonio Vargas Moncada, a mi tio Reynaldo Vargas Nuñez, y a mi querida abuelita Bertita Nuñez, que descansen en la paz del SEÑOR.

A mis padres, hermanos, tíos, primos y demás familiares y amigos por brindarme apoyo para llegar a concluir mi carrera.

A mi tutor Prof. MSc. Ricardo Espinoza y a mi colaborador Prof. Msc. William Noel Martínez por haber depositado en mí su confianza y por haberme guiado durante todo mi trabajo investigativo.

A mi novia Ana Sugey Galeano Bautista, mujer virtuosa que alimento mi mente y mi espiritú, le doy gracias a Dios por tenerla a mi lado y espero tenerla siempre, gracias mi amor por ser para mí un soporte muy fuerte en momentos de angustia y desesperación y por haberme instado en los momentos de flaqueza que tuve a lo largo de este camino.

A mi amiga Ivonne Maria Martínez Montenegro y a mi amigo Rudy Francisco Mairena Ferrufino, personas que desde el primer momento me brindaron y me brindan todo el apoyo, colaboración y cariño sin ningún interés, son las personas por las cuales hoy por hoy puedo afirmar que, a pesar de haber realizado solo mi tesis, jamás me he sentido así, porque ellos han estado a mi lado cada día durante estos años. Ivonne, te quiero mucho y gracias por ser tan linda persona conmigo. Rudy, eres mas que un amigo para mí, eres mi hermano del alma.

A todos los docentes de la carrera de Ingeniería en Sistemas de Información los cuales brindan el pan de la enseñanza para la formación de futuros profesionales.

A todas las personas que a lo largo de mi carrera confiaron en mí y me brindaron su apoyo incondicional, al Sr. César Díaz y a Don Sergio Amaya dos señores que de corazón confiaron siempre en mí.

ÍNDICE GENERAL

Contenido

PROYECTO DE APOYO PARA LA ASIGNATURA: "DISEÑO DE COMPILADORES"

INTRODUCCIÓN

En el presente documento se desarrolla un Proyecto de Apoyo para la asignatura Diseño de Compiladores la cual ofrecen como electiva para el plan de estudio en la carrera de Ingeniería en Sistemas de Información que es ofertada por el Departamento de Computación en la Facultad de Ciencias y Tecnologías de la UNAN-León.

El desarrollo de este Proyecto de Apoyo se planteó en 3 etapas. En la primera etapa se ha definido: los objetivos, la situación de la asignatura en el plan de estudio, la metodología didáctica y material didáctico a utilizar, el proceso de evaluación de la asignatura y la temporización en el semestre. La segunda etapa: consistió en la recopilación de información para desarrollar el contenido teórico previamente definido. En la tercer y última etapa: se elaboraron las propuestas de las prácticas de laboratorios a realizar con su respectiva resolución, la temporización y el sistema de evaluación de dichas prácticas.

El Proyecto de Apoyo está compuesto de dos partes:

TEÓRICA: en esta se desarrolla el contenido teórico del temario planteado y que corresponde a la explicación de la asignatura en clase.

PRÁCTICA: en esta se encuentran las partes de laboratorio que los estudiantes deben seguir, para fortalecer los conceptos teóricos vistos en clase.

Un importante aspecto tomado en cuenta para el éxito del Proyecto de Apoyo es la metodología y técnicas a implementarse en los procesos de enseñanza aprendizaje como una herramienta indispensable para el alcance de la competencia.

Este documento servirá como guía de los elementos claves que se deben impartir en la asignatura de Diseño de Compiladores. El Proyecto de Apoyo deja abierta la posibilidad que el profesor encargado de la asignatura aplique sus propios criterios del curso, haciendo énfasis en aquellos que se consideren convenientes.

OBJETIVOS

Objetivo General

- Desarrollar un Proyecto de Apoyo que sirva como guía de la temática teórica-práctica para la asignatura de Diseño de Compiladores.

Objetivos Específicos

- Elaborar un documento en el que se muestren los aspectos básicos y organizativos de la asignatura Diseño de Compiladores, tales como: situación de la asignatura, metodología y material didáctico, proceso evaluativo, y temporización.

- Incentivar en el estudiante la creatividad, la capacidad de trabajo en grupo y formación autodidáctica a través de la investigación.

- Contar con un documento que sirva como herramienta para el docente encargado de impartir la asignatura así como el alumno que la recibe.

SITUACIÓN DE LA ASIGNATURA EN EL PLAN DE ESTUDIOS EN LA CARRERA DE INGENIERÍA EN SISTEMAS DE INFORMACIÓN DE LA UNAN-LEÓN.

La asignatura Diseño de Compiladores pasó de ser una clase requisito a una clase electiva en el plan de estudio de la Universidad Nacional Autónoma de Nicaragua-León, en la carrera de Ingeniería en Sistemas de Información que ofrece el Departamento de Computación de la Facultad de Ciencias y Tecnologías de dicha Universidad.

La carrera de Ingeniería en Sistemas de Información se desarrolla a lo largo de 10 semestres de estudios, es decir 5 años académicos lectivos para la culminación de la misma. Esta asignatura "Diseño de Compiladores" se imparte una vez que un grupo determinado de estudiantes de la carrera hayan optado a recibir esta asignatura dentro de su plan de estudio como parte de su formación profesional. Anteriormente el periodo de esta asignatura constaba de 6 horas a la semana, de las cuales 4 horas se dedicaban a la parte teórica y 2 horas a la parte práctica. El total de semanas de que consta un semestre es de 16, luego tenemos un total de 32 horas de prácticas y 64 horas teóricas.

El desarrollo y total aprendizaje de la asignatura “Diseño de Compiladores” requiere de unos conocimientos previos adquiridos en otras asignaturas anteriores que facilitarán la comprensión y consolidación de los conocimientos nuevos. Estas asignaturas son:

Programación Estructurada.
Algoritmos y Estructura de Datos.
Programación Orientada a Objetos.

RELACIÓN CON OTRAS ASIGNATURAS

El proceso de formación universitario no consiste en la enseñanza y desarrollo de conocimientos aislados, sino que es un proceso que consta de un conjunto de conocimientos que están estrechamente relacionados, y que para una mejor organización, facilidad de comprensión y aprendizaje se encuentran estructurados y organizados por asignaturas, pero al final todo el conjunto forman parte de los conocimientos básicos y necesarios que cada estudiante debe adquirir para su futura aplicación en el campo profesional. Por tanto no se puede realizar un Proyecto de Apoyo para una asignatura, ni mucho menos impartirla si no se toma en cuenta la relación que existe entre las asignaturas, de forma tal que permita utilizar como base los conocimientos adquiridos previamente y adquirir nuevas bases para las asignaturas posteriores, y para la formación integral del alumno.

En la figura 1.0 se muestran las asignaturas que aportan un conocimiento previo al curso de Diseño de Compiladores.

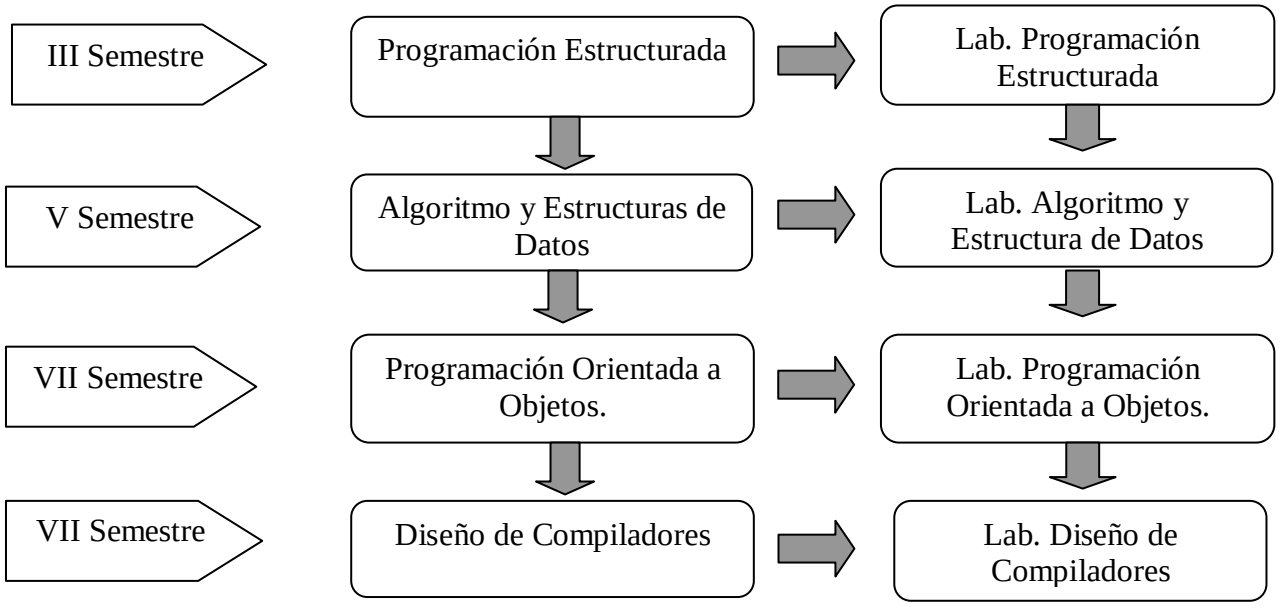
Relación de asignaturas

Fig. 1.0. Relación de asignaturas procedentes a Diseño de Compiladores.

Relación de asignaturas

ASIGNATURA	IMPARTIDA EN
Programación Estructurada	III Semestre
Algoritmos y Estructuras de Datos	V Semestre
Programación Orientada a Objetos	VII Semestre
Diseño de Compiladores	VII Semestre

Una relación más clara entre las asignaturas se muestra a continuación:



Relación Gráfica de Asignaturas Procedentes a Diseño de Compiladores.

Por lo antes mencionado es que se hace necesario mostrar y establecer la relación entre la asignatura de Diseño de Compiladores y las demás asignaturas, relación que se detalla a continuación.

PROGRAMACIÓN ESTRUCTURADA

En esta asignatura se introduce al estudiante en el aprendizaje de las técnicas de programación estructurada y el lenguaje de programación C. Es una asignatura en la que el alumno aprende un lenguaje de programación ya antes mencionado, su aplicación será muy útil a la hora de realizar las prácticas de Diseño de Compiladores, pues la sintaxis del mismo lenguaje "C" y la experiencia de la lógica que se adquiere al programar serán importantes para un buen desarrollo de los laboratorios de Diseño de Compiladores.

En el primer tema se hace una breve introducción a los fundamentos de los ordenadores, haciendo cierto énfasis en los sistemas operativos, programas y lenguajes.

En el segundo tema se abordan las fases en el desarrollo de un programa y los elementos básicos del lenguaje C: tipos fundamentales de datos, tipos derivados, nombres de tipos, constantes, identificadores, expresiones…

En el tercer tema se aborda la estructura de un programa en C: ficheros de cabecera, declaraciones, expresiones, declaración y definición de funciones, sentencias, salida con formato…

El cuarto tema se enfoca en las sentencias de control del lenguaje C: sentencias if, switch, while, for…

El quinto tema aborda los tipos estructurados de datos: arrays, cadenas de caracteres, funciones para manipular cadenas de caracteres, funciones para conversión de datos, estructuras, uniones,…

En el sexto tema se abordan los punteros en C: creación de punteros, operadores de indirección, punteros y arrays, asignación dinámica de memoria…

Hemos llegado al fin de la primera parte de la asiganatura Programación Estructurada, en la segunda parte se continua abordandando elementos más avanzados de la programación en C, para lo cual se necesita tener los conocimientos bases adquiridos en la primera parte.

En el septimo tema se desarrolla la capacidad en el estudiante para que sea capaz de utilizar funciones en C, así como de crear sus propias funciones para su implementación en forma estructurada.

En el octavo tema se aborda las funciones estándar de E/S que proporciona los conocimientos necesarios para que el estudiante sea capaz de utilizar y crear archivos en lenguaje C, y diferenciar entre los diferentes tipos de archivos que se pueden implementar en este lenguaje.

El noveno tema se enfoca en el preprocesador de C: el uso de macros predefinidas y directrices en C en el proceso de precompilación.

El decimo y último tema se centra en el diseño y utilización de estructuras dinámicas de datos y los algoritmos básicos de ordenación.

Todos los conocimientos teóricos abordados en la teoría son llevados a la práctica en los laboratorios.

Esta asignatura es una base importante no solo para Algoritmo y Estructura de Datos que es la continuación de ésta, sino también es de vital importancia para otras asignaturas como son, Programación Orientada a Objetos, Diseño de Compiladores, Programación Visual, Sistemas Operativos, Inteligencia Artificial, y para Redes de Ordenadores, en las cuales se necesita tener conocimientos de programación para poder comprender ciertos elementos teóricos, y para el desarrollo de muchas de las prácticas de sus respectivos laboratorios.

ALGORITMOS Y ESTRUCTURAS DE DATOS

Para esta asignatura es necesario que el alumno tenga una base firme en programación para que pueda comprender más fácilmente la teoría en su totalidad, y pueda desarrollar las prácticas de laboratorio.

En esta asignatura el estudiante aprende a aplicar las técnicas de análisis de los algoritmos, realizar seudocódigos e interpretarlos, evaluar la eficiencia de los algoritmos y a desarrollar algoritmos de búsqueda usando diferentes técnicas.

El primer tema hace una introducción a la algorítmica. Se aborda la representación de los algoritmos, la eficiencia de los algoritmos, el análisis de casos peor y medio entre otras cosas.

El segundo tema aborda las estructuras dinámicas de datos: listas lineales, las operaciones básicas de inserción, borrado de elementos, recorrido de la lista, búsqueda dentro de la lista y el uso de las pilas.

El tercer tema cubre los algoritmos clásicos de recursividad, listas circulares, listas doblemente enlazadas, árboles, recorrido de árboles, árboles de búsqueda, borrado de árboles, y árboles perfectamente equilibrados.

El cuarto tema ahonda en las clasificaciones de datos abordando el método de burbuja, inserción, búsqueda de datos, algoritmo de hash, entre otras cosas.

En esta asignatura también se aplican los conocimientos teóricos en la práctica a través de los laboratorios para los cuales es estrictamente necesario tener conocimientos de programación.

Ésta y la asignatura de programación anteriormente descrita forman una base muy importante e imprescindible para la asignatura de Diseño de Compiladores en la cual se necesita tener fuertes conocimientos de algorítmica y programación para su mayor comprensión y para el desarrollo de las prácticas.

PROGRAMACIÓN ORIENTADA A OBJETOS

Esta asignatura sirve para introducir al estudiante en los conceptos de la programación orientada a objetos, ya que la mayoría de los lenguajes de programación actuales se basan en esta metodología de programación y su conocimiento es de vital importancia para el estudio de otras asignaturas tales como Programación Visual, Diseño de Compiladores, Análisis y Diseño de Sistemas de Información e Ingeniería del Software.

En el primer tema se hace una introducción a la programación orientada a objetos. En este tema se explican los conceptos en los que se fundamenta esta metodología de programación tales como herencia, polimorfismo, etc.

El segundo tema aborda el lenguaje C++ viendo algunos elementos propios del lenguaje y las mejoras con respecto a su antecesor: el lenguaje C.

El tercer tema es uno de los más importantes ya que se encarga de explicar uno de los principales temas: concepto de CLASE.

Este cuarto tema es una continuación del tercero, donde se ahonda aún más en el estudio de las clases, y se ven varios temas de carácter un poco más avanzado tales como miembros de una clase que son punteros, punteros a miembros de una clase, array de objetos, etc.

El quinto tema abarca el tema de OPERADORES SOBRECARGADOS. Se explica cómo sobrecargar los operadores más importantes dentro de un contexto específico, ya sean unitarios o binarios.

El sexto tema aborda otra de las características más importantes de la programación orientada a objetos: el concepto de HERENCIA. También se estudia el concepto de POLIMORFISMO.

En el séptimo tema se estudian los tipos genéricos para comprender la forma de cómo generar plantillas de clase que puedan ser aplicadas a cualquier tipo de datos.

En el octavo y último tema se explica el mecanismo de manejo de excepciones que utiliza C++ para controlar situaciones anómalas durante la ejecución de un programa.

La importancia de esta asignatura en Diseño de Compiladores, se basa en que aporta los conocimientos de la programación orientada a objetos, tan comunes en la codificación de los sistemas modernos y en el diseño de un compilador.

METODOLOGÍA Y MATERIAL DIDÁCTICO

Metodología Didáctica

En esta asignatura de Diseño de Compiladores la metodología a emplear consiste en exposición de las lecciones correspondientes a los temas desarrolados en este proyecto de apoyo en el aula por parte del docente, las cuales se planificarán los temas en clases teóricas de 2 horas de duración. Al finalizar la clase teórica seria bueno hacer siempre un resumen de la clase, este lo puede realizar el profesor con ayuda de los estudiantes para poder evaluar los objetivos planteados en la clase. Luego de haber hecho el resumen de la clase se realizan preguntas de comprobación que se evalúan pero las mismas no se califican, es decir se responde bien, regular o mal; pues acá lo que se está midiendo es si el estudiante captó el contenido nuevo, sirviendo para trazarse los objetivos de la próxima clase. Después de haber hecho las preguntas se realiza un pequeño conversatorio de los contenidos tratados en la próxima clase para que investiguen y en la clase no se encuentren desmotivados por el tema a tratar, es decir se incentiva a los estudiantes a la superación personal, reforzándose en bibliografía a usar en la misma.

Los premios siempre resultan ser motivaciones para el estudiante por lo tanto se les dará 5 pts. extras a los estudiantes que participen activa y voluntariamente en la clase. Otro método a implementar será el de planificar ayuda que se les impartirá a los alumnos que presentan dificultades y las tareas adicionales para los de alto rendimiento para así poner en práctica la bondad, humildad, y el compañerismo.

También se propondrán las prácticas de laboratorio que el alumno desarrollará para que logre un adecuado aprendizaje, y una serie de trabajos y casos de estudios, con los cuales el alumno profundizará sobre el tema, con el objetivo de incentivar su capacidad autodidacta y su capacidad de trabajo en grupo. Todo esto le ayudará a los estudiantes en un futuro cuando tengan que ejercer su profesión.

Material Didáctico Utilizado

Las clases teóricas estarán apoyadas con pizarras acrílicas y DataShow una herramienta importante hoy en día, ya que gracias a esta herramienta, es muy fácil proyectar a los estudiantes, imágenes, esquemas, implementaciones ó resúmenes de aquello que queremos explicar. Debido a que pocos acceden a los folletos o fotocopias por lo que el costo no es muy accesible, el estudiante tendrá a su disposición el tema de teoría y práctica en un archivo de Word, PDF, ó en PowerPoint, el cual será proporcionado por el profesor encargado de la asignatura.

Además de la bibliografía básica se proporcionará al estudiante una bibliografía complementaria que le permitirá profundizar en determinado temas. Los libros de consulta que normalmente son facilitados por el Departamento de Computación ó por la Biblioteca Central. Recursos personales, formados por todos los docentes de la carrera de Ingeniería en Sistemas de Información, ellos constituyen herramientas fundamentales para el desarrollo y enriquecimiento del proceso de enseñanza-aprendizaje de los alumnos. Por favor hagan uso de ellos.

PROCESO DE EVALUACIÓN

La asignatura Diseño de Compiladores está compuesta de dos partes: clases teóricas y prácticas de laboratorios**.**

Evaluación de la parte teórica:
Durante el transcurso del semestre se realizarán **3** evaluaciones parciales y finalmente estas serán sumadas y divididas entre el número de parciales. Las calificaciones obtenidas en las **3** evaluaciones parciales equivalen al **60%** de la nota final teórica, el restante **40%** de la nota final teórica lo representan las prácticas de laboratorios.

Evaluación de las prácticas de laboratorios:
Durante el semestre se realizarán **3** evaluaciones parciales de las prácticas de laboratorios coincidiendo con los períodos de evaluación de la teoría, estas evaluaciones podrán ser pruebas, defensas de proyectos, o una combinación de estas. Lo que tendrá un valor de **40 pts**. cada una de estas evaluaciones.

Un resumen de lo antes mencionado puede verse en la siguiente tabla:

	Primera Evaluación Parcial	**Segunda Evaluación Parcial**	**Tercera Evaluación Parcial**	**Nota Final**
Teoría	60	60	60	60
Prácticas	40	40	40	40
Total	100	100	100	**100**
%	33.33%	33.33%	33.33%	**100%**

Fig. 1.1. Tabla de proceso de evaluación.

Cálculo de la **Nota Final**

EIP= 0.6 * EIPT + 0.4 * EIPPL
EIIP = 0.6 * EIIPT + 0.4 * EIIPPL
EIIIP= 0.6 * EIIIPT + 0.4 * EIIIPPL

Nota Final = (EIP + EIIP + EIIIP)/3

Donde:

EIP es la Evaluación del I Parcial.
EIIP es la Evaluación del II Parcial.
EIIIP es la Evaluación del III Parcial.
EIPT es el Examen del I Parcial Teórico.
EIPPL es la Evaluación del I Parcial de las Practicas Laboratorios.
EIIPT es el Examen del II Parcial Teórico.

EIIPPL	es la Evaluación del II Parcial de las Practicas Laboratorios.
EIIIPT	es el Examen del III Parcial Teórico.
EIIIPPL	es la Evaluación del III Parcial de las Practicas Laboratorios.
Nota Final	es la nota definitiva del estudiante.

Aspectos especiales de la evaluación:

- Si la calificación final del estudiante (**Nota Final**) es mayor de **59** (**Aprobado**), y si es menor de **60** el estudiante tiene derecho a una nueva oportunidad de presentarse a una segunda convocatoria. En ésta, su nota definitiva es la calificación que obtenga en dicha convocatoria, es decir, que para efectos de ésta, las calificaciones anteriores no se toman en cuenta.

PLANIFICACIÓN TEMPORAL

La carrera de Ingeniería en Sistemas de Información se desarrolla a lo largo de 10 semestres de estudios, es decir 5 años académicos lectivos para la culminación de la misma. Esta asignatura "Diseño de Compiladores" se imparte en dicha carrera, la cual es ofrecida por el Departamento de Computación de la Facultad de Ciencias y Tecnologías de la UNAN León. Cada semestre consta de 16 semanas lectivas. La asignatura tiene una frecuencia de 6 horas semanales (4 de teoría y 2 de prácticas de laboratorios), resultando en un total 64 horas teóricas y 32 horas prácticas para un total de 96 horas.

Si consideramos los días festivos, las semanas de realización de exámenes y dando margen a otras incidencias la cantidad de semanas por semestre se reducen a 14, obteniendo ahora **56 horas de teoría** y **28 horas de laboratorio** respectivamente.
Hay que recordar que cada sesión de clase teórica dura 2 horas reloj; así que por semanas hay 2 sesiones teóricas de clases. La sesión de laboratorio también tiene una duración de 2 horas reloj; por lo que cada semana solamente hay 1 sesión de laboratorio.

CONTENIDO DEL TEMARIO

Tema 1: INTRODUCCIÓN

- Reseña histórica de los compiladores.
- Conceptos básicos.
- Estructura de un compilador.

Tema 2: ESQUEMA GENERAL DE UN COMPILADOR

- Estructura de un compilador para SLANG AMPLIADO.
- Scanner para SLANG AMPLIADO.
- Sintaxis para SLANG AMPLIADO.
- Parser recursivo descendente.
- Traduciendo SLANG AMPLIADO.

Tema 3: SCANNER: TEORÍA Y PRACTICA:

- Expresiones regulares.
- Autómatas finitos y scanners traduciendo Expresiones regulares y Autómatas finitos.
- Consideración práctica.
- Generador de Scanners: FLEX

Tema 4: GRAMÁTICA LIBRE DE CONTEXTO Y PARSER:

- Conceptos y notación.
- Errores comunes en la Gramática Libre de Contexto la Forma Bachus Naur Extendida..
- Parser y Reconocedores.
- Algoritmo Analizador de Gramática.

Tema 5: TÉCNICA DE PARSEO LR Y LL:

- Técnica de Parseo LL y sus elementos.
- Bison, Generadores de Parser.
- La técnica de Parseo LR y sus elementos.

DESARROLLO DEL TEMARIO

En esta sección se desarrollan los temas a impartir en la asignatura **Diseño de Compiladores.** Se comienza a partir del tema cero, que es una presentación de la asignatura, luego se prosigue con el desarrollo de los temas.

Tema 0: "PRESENTACIÓN DE LA ASIGNATURA"

Objetivos

- Dar a conocer al estudiante sobre los aspectos relacionados a la docencia en general y horario de tutorías.
- Explicar los objetivos generales de la asignatura, el temario y su distribución en el tiempo, así como las relaciones de la misma con la carrera: Ingeniería en Sistemas de Información.
- Presentar al estudiante la metodología de enseñanza que se va a seguir durante el curso, los criterios de evaluación y la bibliografía básica y complementaria.
- Brindar detalles del acceso a la documentación preparada por el profesor para impartir el componente.

Contenido

- Objetivos de la asignatura.
- Contenido de la asignatura.
- Temporización.
- Metodología de evaluación
- Bibliografía a utilizar.
- Horarios de tutorías.

Objetivos de la asignatura.

Se plantearán los objetivos que se pretenden alcanzar durante el desarrollo de la asignatura, tanto en la parte práctica como teórica.

Contenido de la asignatura.

Se describirá de forma general cada uno de los temas contemplados en el temario antes presentado, así como también de las prácticas a desarrollar en el laboratorio.

Temporización.

Se expondrá al alumno, la temporización establecida para la asignatura, tanto en su parte teórica, como es su parte práctica, desglosando el tiempo, con el que se cuenta para cada uno de los temas y prácticas que se cubrirán a lo largo del semestre.

Metodología de evaluación.

Antes de comenzar a desarrollar la asignatura, es preciso dar a conocer a los estudiantes los criterios de evaluación a implementar en la asignatura así como también aclarar la forma de evaluar las actividades propuestas por el docente.

Bibliografía a utilizar.

Se dará conocer al alumno la bibliografía básica que se utilizará para el desarrollo de la asignatura, así como también los diferentes medios de acceso que tiene a ella.

Horarios de tutorías.

Todos los alumnos, tendrán el derecho de hacer consultas al profesor, fuera del aula de clases en un determinado horario, que el profesor debe hacer del conocimiento del alumnado.

Tema 1: INTRODUCCIÓN

Objetivos

- Narra de forma breve reseña histórica del surgimiento de los compiladores.
- Define conceptos básicos de compilador.
- Identifica la estructura de un compilador.

Reseña histórica de los compiladores

El término de compilador surgió en los inicios de 1950 por idea de Grace Murray Hopper. Como ejemplo de los primeros compiladores en el sentido moderno tenemos el FORTRAN, el cual surgió a finales de 1950. Para aquellos entonces la compilación era vista como un secuencia de subprogramas seleccionados de una biblioteca.

Generalmente hablando, un compilador es un programa que lee un programa escrito en un lenguaje fuente y lo traduce en un programa equivalente en un lenguaje destino. El lenguaje destino de un compilador generalmente necesita futuro procesamiento antes de que este pueda ser ejecutado.

¿Porqué los compiladores?

Cada sistema de computadora posee un lenguaje de máquina. Los programas escritos en lenguaje de máquina pueden ser ejecutados por el hardware del sistema, esto es, el hardware es capaz de descodificar las instrucciones del lenguaje de máquina y realizar las acciones de acuerdo a sus significados. El hardware del sistema ejecuta solamente programas escritos en el lenguaje de máquina. Pero existen programas que no son escritos en lenguaje de máquina, asumamos que tienen lenguajes de alto nivel como C o Pascal, estos programas son lenguajes de alto nivel (*printf lenguaje humano*) y no pueden ejecutarse por el software del sistema.

Para propósitos prácticos tomaremos un lenguaje de alto nivel llamado SLANG AMPLIADO para desarrollar la teoría y la práctica de compiladores. Si una computadora pudiera ejecutar directamente un programa escrito en SLANG AMPLIADO esto significa que el SLANG AMPLIADO es su lenguaje de máquina. Desdichadamente, una maquina de este tipo no existe.

1. **Una solución es desarrollar un programa intérprete que lea e interprete (No traduce a lenguaje de máquina) las órdenes del programa escrito en SLANG AMPLIADO, esto es, reconocer cada orden y realizar las acciones asociadas. De hecho, un programa intérprete simula una máquina SLANG AMPLIADO.**

Un ***intérprete***, difiere de un compilador en que este ejecuta programas sin explicita presentación de una traducción. A continuación la organización de un intérprete ideal.

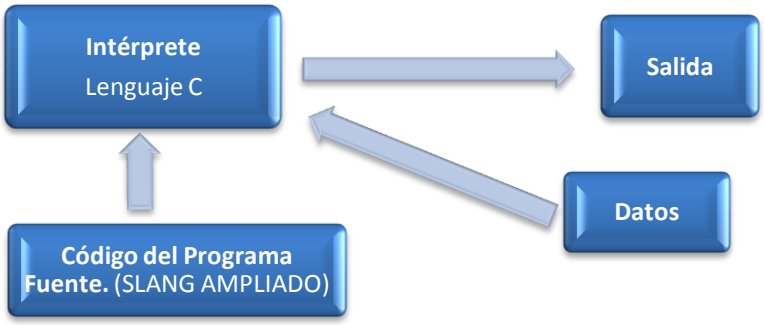


Fig. 1.2. Un itérprete.

Los intérpretes permiten lo siguiente:

- Modificación de o adición a programas del usuario a medida que la ejecución procede.
- Lenguajes en los cuales el tipo de objeto que una variable denota puede cambiar dinámicamente.
- Mejores diagnósticos.
- Un significante grado de independencia de la máquina puesto que no hay código de máquina generado.

Sin embargo, la interpretación puede envolver grandes dificultades:

- Velocidad de ejecución de 10:1 a 100:1 en el factor de degradación.
- Utilización de gran cantidad de espacio en memoria ya que el intérprete y todas sus rutinas deben estar disponibles siempre.

Algunos lenguajes como BASIC, LISP y Pascal, tienen ambos intérpretes (para depuración y desarrollo de programas) y compiladores (para trabajo de producción).

2. **Otra solución es primero transformar (traducir) el programa en SLANG AMPLIADO en un programa equivalente en un lenguaje para el cual el proceso de interpretación sea más fácil y después tener otro programa intérprete para este lenguaje que ejecute las instrucciones del programa traducido (Este se traduce al lenguaje de máquina).**

El lenguaje de máquina es el lenguaje más fácil de interpretar porque para interpretar programas en este lenguaje no se necesitan herramientas software.

El problema de diseñar un traductor para el lenguaje SLANG AMPLIADO es expresar el significado de cada programa SLANG AMPLIADO en instrucciones de máquina. Debido a que SLANG AMPLIADO es un lenguaje de alto nivel, la distancia entre SLANG AMPLIADO y los lenguajes de máquina es grande. Por lo tanto, un lenguaje intermedio es siempre utilizado en el proceso de traducción. Este lenguaje intermedio nos permite realizar la traducción no en un paso largo sino en dos pasos pequeños.

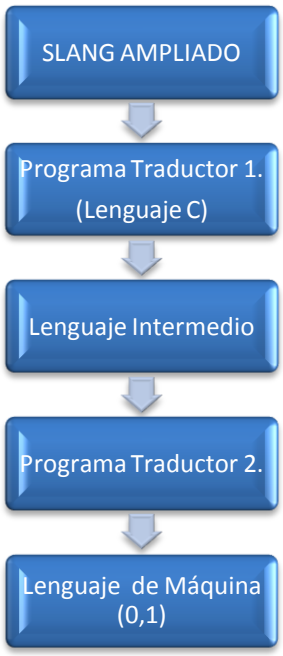


Fig. 1.3. Proceso de traducción para SLANG AMPLIADO.

Una vista relacionada con traducción es el modelo de análisis y síntesis. La fase de análisis reconoce la estructura y el significado del programa fuente, mientras que la fase de síntesis construye el programa destino deseado.

Los compiladores son esenciales para la computación moderna. Ellos actúan como traductores, transformando lenguajes de programación humano-orientado en lenguaje de máquina computador-orientado. La mayoría de los usuarios pueden ver un compilador como una "caja negra". Un traductor que realiza la traducción de un lenguaje de alto nivel en lenguaje intermediario o un lenguaje maquina es llamado **compilador.**

Los Compiladores: Un traductor que realiza la traducción de un lenguaje de alto nivel (lenguaje de programación humano) en un lenguaje intermediario ó lenguaje máquina (computador).

El lenguaje destino de un compilador generalmente necesita futuro procesamiento antes de que este pueda ser ejecutado.

Fig. 1.4. Un compilador.

Clasificación de los Compiladores

- Compiladores de Diagnostico: son diseñados especialmente para ayudar en el desarrollo y depuramiento de programas. Estos tienen la capacidad de incluir chequeo de código que pude detectar errores en tiempo de corrida y abortar la ejecución de un programa. En este caso no omitimos expresar que estamos construyendo un traductor de una sola pasada (lenguaje C).

- Compiladores Optimizadores: son especializados para producir código de máquina eficiente, con el costo que los compiladores crecen en complejidad y tiempo de compilación.

Los Compiladores permiten que los programas y los expertos en programación sean "máquinas-independientes".

En la compilación, el análisis consta de tres fases:

1. **Análisis lineal o léxico,** en el que la cadena de caracteres (BEGING) que constituyen el programa fuente se lee de izquierda a derecha y se agrupa en componentes léxicos (TOKEN "BEGING_"), que son secuencias de caracteres que tienen significado colectivo. A esto se le conoce como **SCANNER.**

2. **Análisis jerárquico o sintáctico,** en el que los caracteres o los componentes léxicos (TOKEN) se agrupan jerárquicamente en colecciones anidadas **(READ(var);)** con un significado colectivo. A esto se le conoce por **PARSER.**

3. **Análisis semántico o de contexto,** en el que se realizan ciertas revisiones para asegurar que los componentes de un programa se ajustan de un modo significativo. Ellas definen los detalles como cada construcción debe ser chequeada y traducida.

En esta fase se hace un árbol sintáctico abstracto determinado por el PARSER. Muestra las relaciones entre los operadores y sus operandos en las expresiones. También verifica las asignaciones de los tipos de datos. Si la contracción es semánticamente correcta también realizan la actual traducción (Código IR o representación intermedia) que implementa correctamente la construcción generada.

A continuación mostramos el sistema para el procesamiento de un lenguaje:

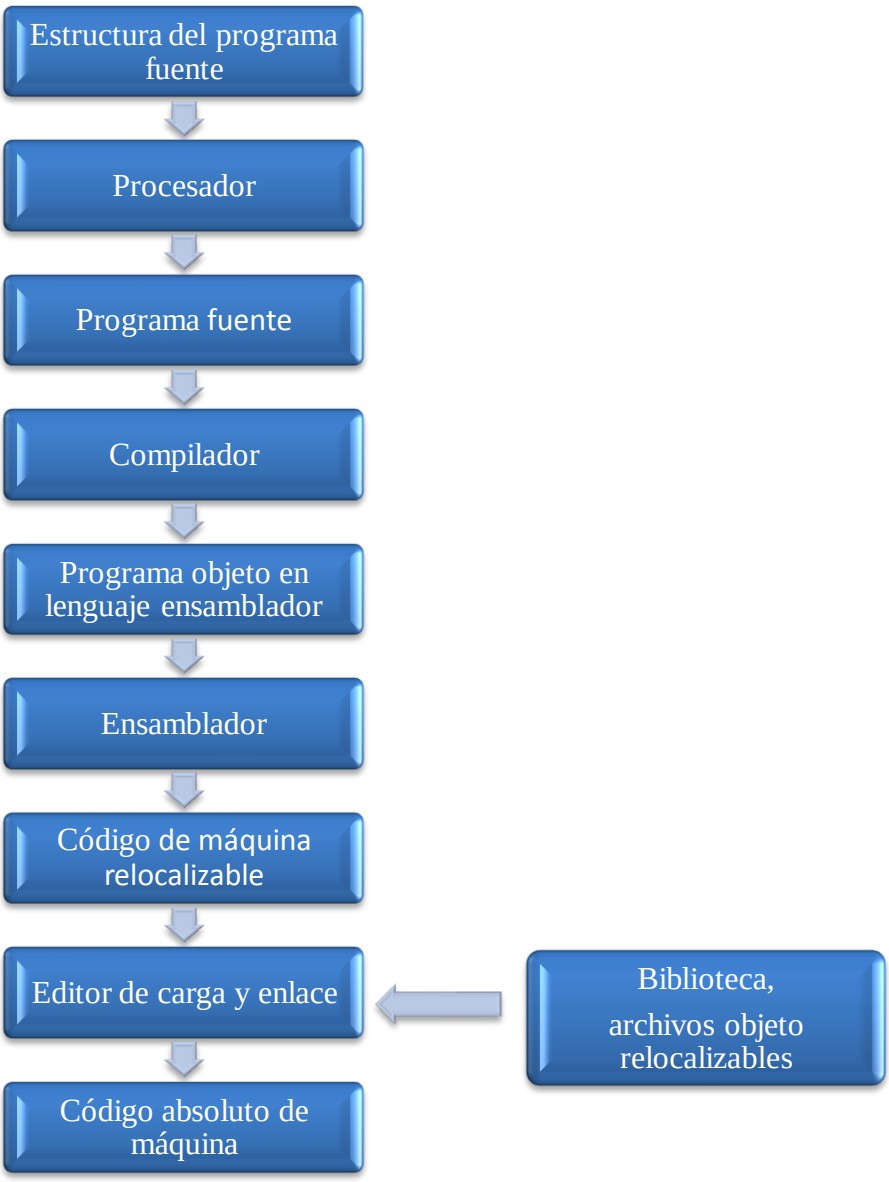


Fig. 1.5. Sistema para procesamiento de un lenguaje.

Los procesadores producen la entrada para un compilador, y pueden realizar las funciones siguientes:

- Procesamiento de macros
- Inclusión de archivos
- Preprocesadores “racionales” (flujo de control y estructuras de datos)
- Extensiones a lenguajes

Algunos compiladores producen código ensamblador, otros realizan el trabajo ensamblador, produciendo código de máquina relocalizable que se puede pasar directamente al editor de carga y enlace. El código ensamblador es una versión nemotécnica de código de máquina, donde se usan nombres en lugar de códigos binarios para operaciones y también se usan nombres para las direcciones de memoria.

Un programa llamado cargador, realiza las dos funciones de carga y edición de enlace. El proceso de carga consiste en tomar el código de máquina relocalizable, modificar las direcciones relocalizable y ubicar las instrucciones y los datos modificados en las posiciones apropiadas de la memoria. El editor de enlace permite formar un solo programa a partir de varios archivos de código de máquina relocalizable.

La estructura de un compilador

Cualquier compilador debe ejecutar dos grandes tareas: **análisis** del programa origen a ser compilado y **síntesis** de un programa de lenguaje de máquina que, cuando sea ejecutado, ejecutará correctamente las actividades descritas en el programa origen.

La estructura de un compilador sintáctico-dirigido:

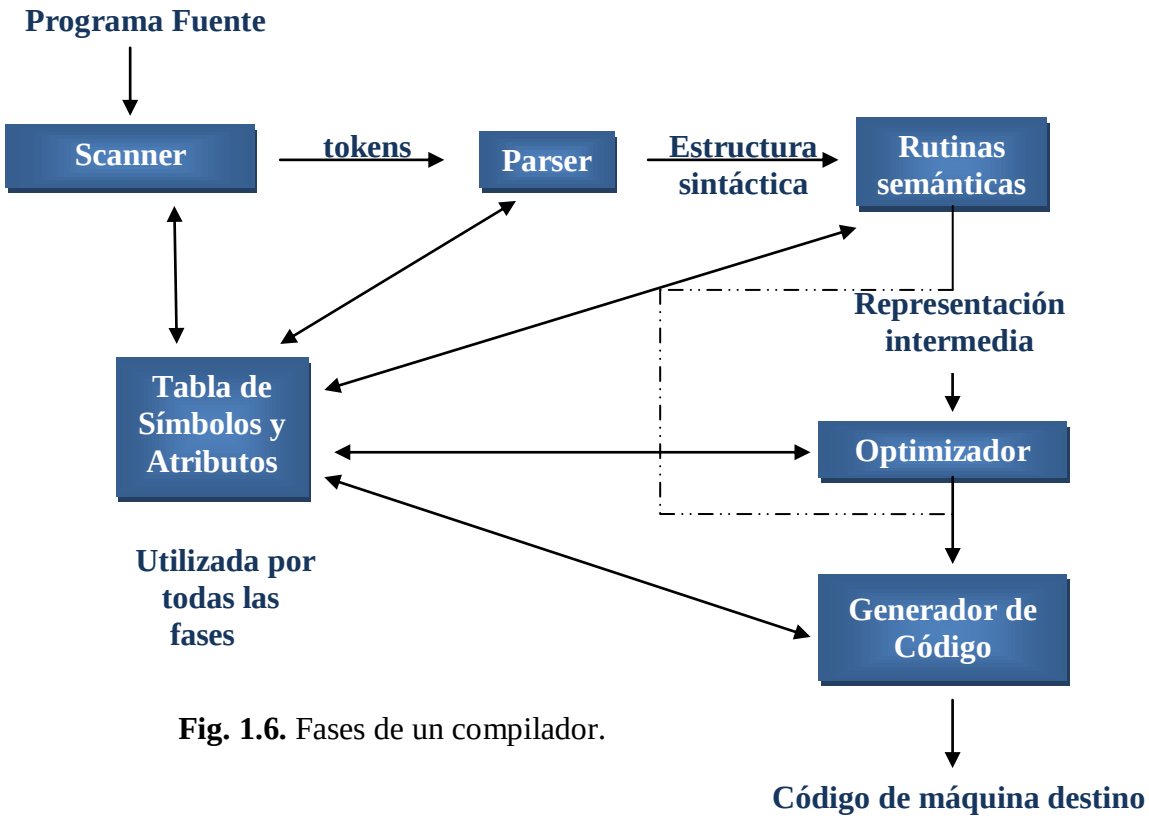


Fig. 1.6. Fases de un compilador.

El scanner comienza con el análisis del programa origen, leyendo la entrada, carácter por carácter, y agrupando caracteres en palabras individuales y símbolos. Las palabras son llamadas tokens y representan entidades básicas del programa tales como identificadores, enteros, palabras reservadas, delimitadores, etc. Este es el primero de varios pasos que produce sucesivas interpretaciones de alto nivel de la entrada. Los tokens son codificados y son enviados al parser para el análisis sintáctico.

El scanner hace lo siguiente:

- Transforma el programa en un formato compacto y uniforme (tokens).
- Elimina información innecesaria (tales como comentarios).
- Procesa directivos de control del compilador a través de una lista de pseudo componentes.
- Introduce información preliminar en la tabla de símbolos y atributos.
- Formatea y lista el programa objeto.

Dada una sintaxis formal especificada (típicamente una gramática libre de contexto (CFG)), el parser lee los tokens y los agrupa en unidades como las especifican las producciones de la gramática empleada. El parser verifica la sintaxis correcta, y si un error sintáctico es encontrado, el parser emite un diagnostico.

Las ***rutinas semánticas*** realizan dos funciones. La primera, chequear la semántica estática de cada construcción. Esto es, verificar que la construcción es legal y completa en significado. Si la construcción es semánticamente correcta, las rutinas semánticas también realizan la actual traducción. Esto es, código IR (de representación intermedia) que implementa correctamente la construcción que es generada. El corazón del compilador descansa en las rutinas semánticas ya que ellas definen los detalles de cómo cada construcción debe ser chequeada y traducida.

El código generado por las rutinas semánticas es analizado y transformado en código IR funcionalmente equivalente pero mejorado por el ***optimizador*****.** Esta fase es bien compleja y lenta en su ejecución; usualmente implica un gran número de subfases algunas de las cuales pueden ser necesitadas para ser aplicadas más de una vez.

El ***generador de código*** genera lenguaje de máquina a partir del código IR recibido del optimizador. Este requiere información del código de máquina destino y generalmente envuelve aspectos de optimizaciones de especificaciones de máquina, tales como alocución de registro, escoger el formato de las instrucciones, modelo de direccionamiento.

La ***tabla de símbolos y atributos*** es un mecanismo que permite que la información (atributos) sea asociada con los identificadores. Estas tablas pueden ser usadas por cualquier componente del compilador para compartir y recuperar información acerca de las variables, procedimiento, etiquetas y así sucesivamente.

Detección e Información de Errores

Cada fase puede encontrar errores y detener el proceso de compilación enviando un mensaje. Sin embargo, después de detectar un error, cada fase debe tratar de alguna forma ese error, para poder continuar la compilación, permitiendo la detección de más errores en el programa fuente e informando de todos ellos en el proceso de interacción con el usuario. Un compilador que se detiene cuando encuentra el primer error no resulta tan útil como debiera.

Clasificación de los Compiladores

- **Compiladores de Diagnósticos:** que son diseñados especialmente para ayudar en el desarrollo y depuración de programas. Estos tienen la capacidad de incluir chequeo de código que pueden detectar errores en tiempo de corrida y abortar la ejecución de un programa.
- **Compiladores Optimizadores:** que son especializados para producir código de máquina eficiente, con el costo que los compiladores crecen en complejidad y tiempo de compilación.

La Sintaxis y la Semántica de los Lenguajes de Programación

Una completa definición de un lenguaje de programación debe incluir especificaciones de su sintaxis (estructura) y de su semántica (significado).

Considere una sintaxis para una orden en C

expr → expr operador expr
expr → (expr)
expr → **constante**
expr → **identificador**
operador → + | - | * | / (| alternante)

La categoría sintáctica expr denota una notación para que pueda ser definido el conjunto de todas las cadenas de caracteres que satisfacen la definición de expresión.

La definición del lenguaje descrito anteriormente es un generador de esquemas en el sentido que cualquier orden bien formada puede ser generada comenzando en la categoría sintáctica expr y aplicar sucesivamente las otras reglas. Esto puede ser explicado en una estructura de árbol donde el nodo raíz corresponde con el símbolo de inicio y cada nodo es etiquetado por un símbolo de la gramática. Un nodo interior y sus hijos corresponden a una regla; el nodo interior corresponde al lado izquierdo de la producción y los hijos corresponden al lado derecho.

Por ejemplo:
Ejemplo 1.0. El árbol sintáctico de la expresión (5+3) – (8 * 5) es:

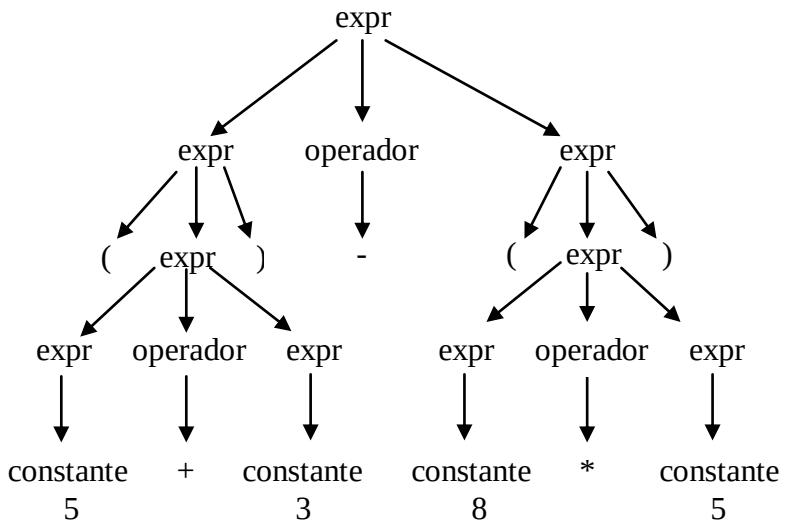


Fig. 1.8. El árbol de análisis sintáctico para (5+3) – (8 * 5).

Existen dos tipos de símbolos: **terminales** y **no terminales.**

Los símbolos no terminales: pueden ser reconocidos por el hecho de que ellos aparecen al lado izquierdo de las producciones.

Los terminales: nunca cambian. Ellos representan tokens del lenguaje.

De esta manera, el propósito general de un conjunto de producciones (CFG) es el de especificar que secuencia de terminales (tokens) son legales. El símbolo λ representa el vacío o la cadena vacía. Así una producción del tipo A → λ expresa que A puede ser remplazada por la cadena vacía. La estructura así como la sintaxis pueden ser definidas en una CFG. Para expresiones, esto incluye las reglas de asociatividad y precedencia de los operadores.

En nuestro ejemplo son símbolos terminales: constante, identificador, +, -, *, /, (, y); son símbolos no terminales: expr y operador.

Gramáticas Libres de Contextos: Conceptos y Notación:

Una CFG está definida por los cuatro siguientes componentes:

1. Un vocabulario terminal finito Vt; esto es el conjunto de tokens producido por el scanner.

2. Un conjunto de diferencia finito, símbolos intermedios, llamado el vocabulario no terminal Vn.

3. Un símbolo de inicio $S \in Vn$ que inicia todas las derivaciones.

4. Un conjunto finito de producciones P (algunas veces llamado reglas de reescritura) de la forma

   $A \rightarrow X_1 \ldots X_m$, donde $A \in V_n$, $X_i \in V_n \cup Vt$, $1 \leq m$, $m \geq 0$.

   Note que $A \rightarrow \lambda$ es una producción válida.

   Abreviadamente:
   CFG = (Vt, Vn, S, P)

El vocabulario V de una CFG es el conjunto $Vt \cup Vn$.

El conjunto de cadenas derivables de S define el lenguaje libre de contextos de la gramática G, denotado por L(G).

Una gramática genera o deriva una orden comenzando por el símbolo de inicio y aplicando repetidamente una producción. La aplicación de las producciones consiste en el reemplazo del lado izquierdo por una de sus alternativas en el lado derecho, el cual puede ser incluso vacío.

El árbol del ejemplo anterior es llamado **árbol de parseo** y demuestra como una cadena de caracteres en el lenguaje puede ser derivada desde el símbolo de inicio. Formalmente un árbol de parseo para una gramática libre de contexto tiene las siguientes propiedades:

1. La raíz es etiquetada por el símbolo de inicio.
2. Cada hoja es etiquetada con un token o λ.
3. Cada nodo interior es etiquetado con un símbolo no terminal. Si X_0 es el no terminal etiquetando un nodo y X_1, $X_2 \ldots$, X_n, son las etiquetas de sus hijos de izquierda a derecha, entonces debe existir una producción $X_0 \rightarrow X_1 X_2 \ldots X_n$, donde, a su vez, cada X_i puede ser un símbolo terminal o no terminal. Si el nodo tiene un solo hijo etiquetado por λ, entonces es que existe la producción $X_0 \rightarrow \lambda$.

Ambigüedad

Una cadena de caracteres es llamada ambigua si dicha cadena tiene más de un árbol de parseo.

No siempre es posible hablar acerca de árbol de parseo de una cadena de caracteres acuerdo con una gramática ya que dicha cadena puede tener más de un árbol de parseo. Tal cadena de caracteres es llamada ambigua.

Una gramática es ambigua si de ella se pueden derivar cadenas de tokens que son ambiguos. De aquí que para demostrar que una gramática es ambigua simplemente debemos encontrar una cadena de caracteres que tenga más de un árbol de parseo.

Las gramáticas ambiguas no son prácticas y por tanto siempre se debe comprobar que una gramática no es ambigua antes de diseñar un compilador para el lenguaje que define.

Ejemplo 1.1:

expr → expr + expr | expr – expr | digit
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Asociatividad de los operadores

Considere la expresión 3 – 2 + 1. Como el operando 2 tiene operadores a su izquierda y a su derecha, una regla para evitar la ambigüedad es necesaria para decidir cual operador toma cual operando. Por convención 3 – 2 + 1 es equivalente a (3 – 2) + 1. Por esto se dice que los operadores +, -, *, y / se dicen que son asociativo por la izquierda mientras que la exponenciación y la asignación son asociativa por la derecha.

Para resolver la ambigüedad en la gramática previa podemos reescribir las producciones de la siguiente forma:

expr → expr + **digit** | expr – **digit** | digit
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

El operador de exponenciación puede ser expresado como:

expr → digit ↑ expr | digit
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

La diferencia entre los árboles de parseo para operadores asociativos por la izquierda y asociativos por la derecha es que el primero crece hacia abajo por la izquierda mientras el segundo crece hacia abajo por la derecha.

Precedencia de Operadores

Ahora consideremos la combinación entre suma y multiplicación en una expresión. La expresión 3 + 2 * 1, tiene dos posibles interpretaciones, estas son, (3 + 2) * 1 y 3 + (2 * 1). La asociatividad de + y de * no resuelven este tipo de ambigüedad. Por lo tanto necesitamos conocer la precedencia relativa de los operadores + y * cuando ellos tienen los mismos operandos.

Por conveniencia, el operador * toma sus operandos antes que el operador + ya que el operador * tiene mayor precedencia que +. La expresión 3 + 2 * 1 es equivalente a 3 + (2 * 1).

Ejercicio: Construir una gramática para expresiones aritméticas dada la tabla siguiente:

Operador	Asociatividad
↑	Derecha
* y /	Izquierda
+ y -	Izquierda

Restricciones de Contexto

Hasta el momento hemos visto que un lenguaje puede ser definido por medio de una gramática libre de contexto. En muchos lenguajes de programación, sin embargo, aspectos sensitivos de contextos, por ejemplo: el alcance y las reglas de tipo, también juegan un papel muy importante. **Las reglas de alcance** determinan el alcance de cada declaración y permite la declaración de cada identificador a ser localizado. ***La regla de tipo*** hace posible inferir el tipo de cada expresión y así asegurar que cada operador es proveído por los operandos del tipo correcto.

Ejemplo de regla de alcance:

```
int numero[30];    /*Alcance de cada declaración*/
float var;         /*declaración de id a ser localizado en memoria al correr el programa*/
```

Análisis del programa fuente

El *análisis léxico (SCANNER)* recordemos que traduce la cadena de caracteres de un programa fuente en una serie de tokens donde cada token corresponde a una subcadena. Por ejemplo la secuencia de caracteres en la orden de asignación:

Ejemplo 1.3. Fahrenheit: = 32 + Celsius * 1.8

Sería traducida en la siguiente secuencia de tokens:

- El identificador Fahrenheit
- El operador de asignación: =
- El entero 32
- El operador de la suma +
- El identificador Celsius
- El operador de multiplicación *
- El real 1.8

Los **espacios en blanco** entre los tokens usualmente son eliminados en la primera fase del análisis por el scanner, así el parser no tendrá que considerarlos. Los comentarios pueden ser tratados como espacios en blanco ya que no tienen ningún significado para el código del programa.

Las **cadenas que corresponden a números** por ejemplo puede ser en la declaración de las variables al inicializar números reales o enteros son reconocidas por el scanner y convertidas en un token. Después el analizador léxico pasará los tokens integer o real y sus valores asociados al parser. El parser podrá después tratar los números como unidades simples.

Cuando el scanner reconoce un **identificador (cadena de caracteres)** en la entrada, se necesita algún mecanismo para determinar cuando la cadena de caracteres correspondientes ha sido antes vista. La cadena es entonces almacenada en una tabla de símbolos y atributos, si este identificador no ha sido incluido con anterioridad.

Todo lenguaje tiene **palabras claves** tales como begin, end, if, then, etc. para marcar el inicio de alguna construcción así como también las marcas de puntuación como punto y coma, punto, paréntesis, etc. Estas palabras claves se tratan como identificadores reservados que son en algunos casos incluidos en la tabla de símbolos y atributos (TSA) y marcados como palabras claves.

Dada un cadena de tokens y una gramática, el parseo o análisis sintáctico es el proceso de determinar si y como la cadena de tokens puede ser generada por la gramática. Aún cuando el compilador no construya un árbol de parseo, es siempre de gran ayuda imaginar que el árbol de parseo está siendo construido.

La mayoría de los métodos de parseo trabajan de arriba hacia abajo o de abajo hacia arriba. Estos términos se refieren al orden en el cual los nuevos nodos del árbol de parseo son agregados durante el proceso de su construcción.

Explicaremos brevemente como funciona un parseo de arriba hacia abajo por medio de una gramática simplificada para expresiones:

assignment → identifier := expr
expr → expr + term | term
term → term * factor | factor
factor → (expr) | identifier | integer | real

Ejemplo 1.4.
Construimos el árbol de parseo

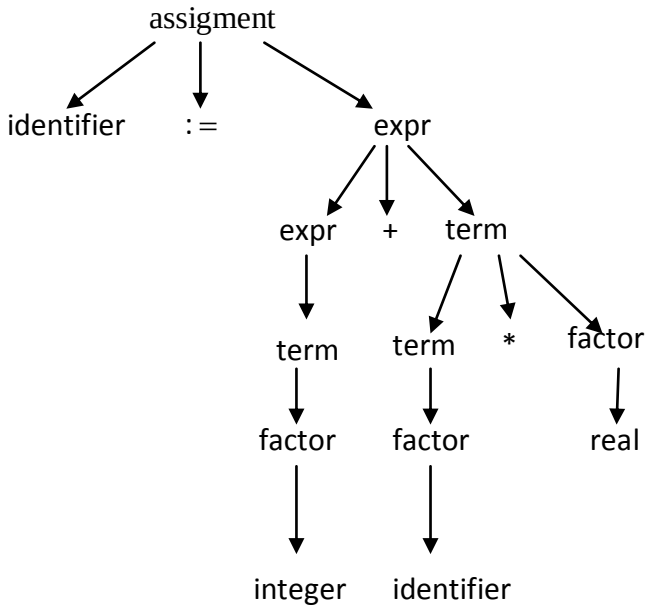


Fig. 1.10. Árbol de parseo de arriba hacia abajo.

El análisis de contexto verifica si el programa fuente satisface todas las restricciones y recolecta toda la información sobre tipos para la fase de síntesis de la compilación. En esta fase se utiliza un árbol sintáctico abstracto determinado por el analizador sintáctico, el cual muestra las relaciones entre los operadores y sus operandos en las expresiones.

Una tarea importante de un analizador de contexto es la verificación de tipos, por ejemplo, congruencia de tipos, el índice de un arreglo debe ser siempre real, los operandos de un operador boolean, deben ser de tipo boolean, etc.

Síntesis del programa destino

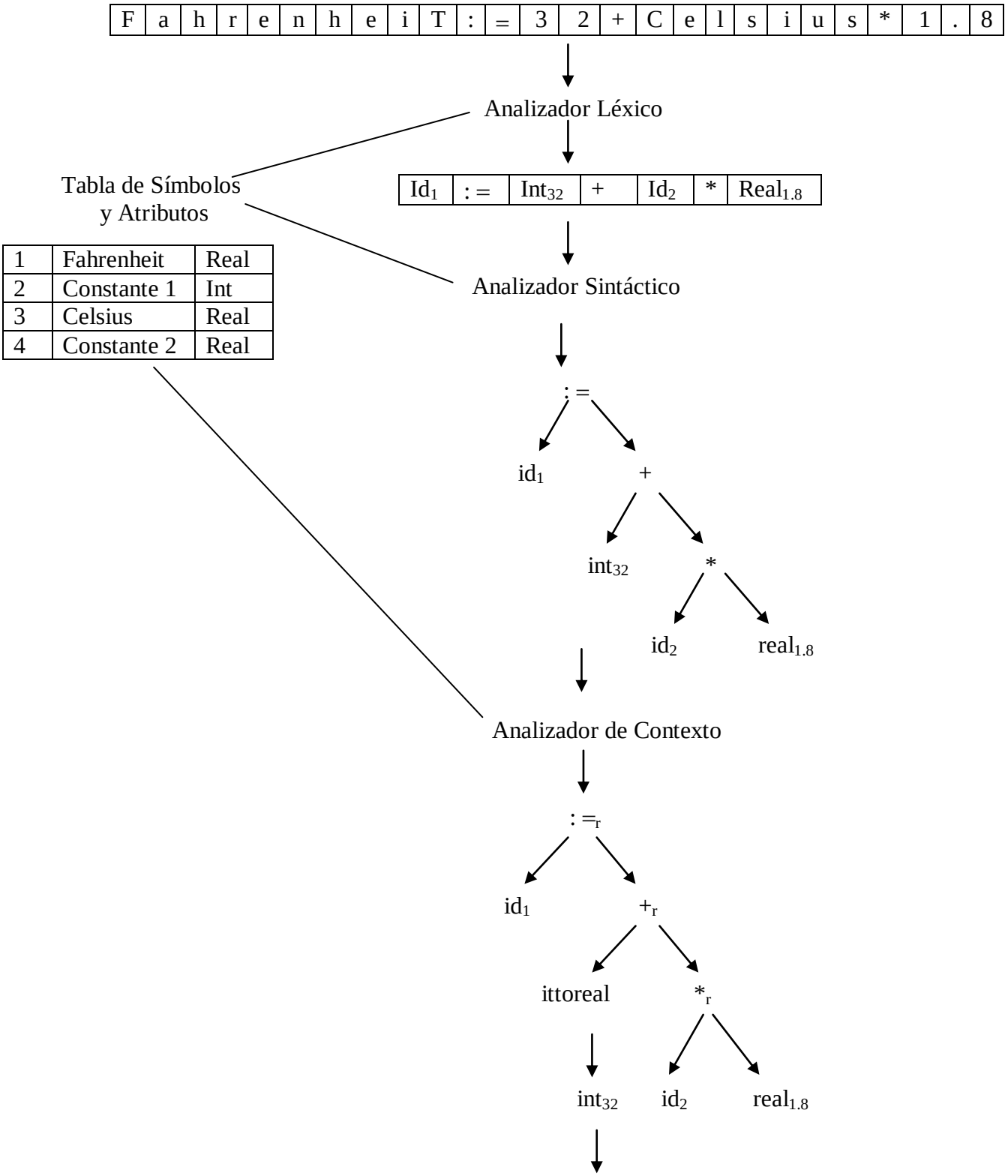
Conceptualmente, la síntesis consiste en tres partes:

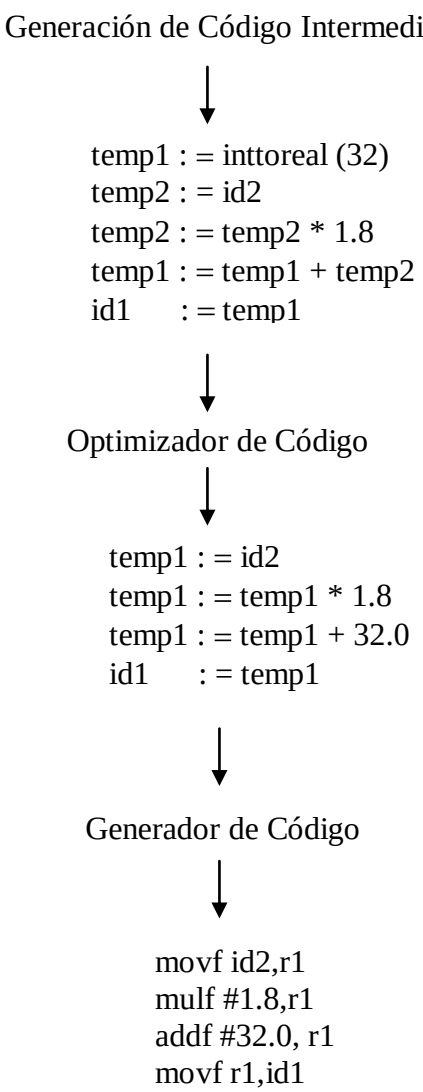
1. Generación de código intermedio
2. Optimización de código
3. Generación de código

La generación de código intermedio requiere la traducción del programa original en una representación intermedia, esta representación intermedia es primero optimizada y luego el código de máquina es generado.

En la siguiente figura mostraremos la aplicación de todas las fases de análisis y síntesis a la orden

Fahrenheit:= 32 + Celsius * 1.8





En el optimizador de código se redujo una línea de IR recuerde que puede repetir este paso hasta que este al máximo de optimizado luego vamos al generador de código donde están las sentencias en el ensamblador o lenguaje de máquina.

Fig. 1.11. Fases de análisis y síntesis a la orden Fahrenheit: = 32 + Celsius * 1.8

Traductor de una sola pasada

A como hemos explicado, un compilador trabaja en fases, cada una de las cuales transforma el programa original de una representación a otra. En la práctica, sin embargo, algunas fases pueden ser combinadas en una sola. Esta pasada (usualmente) consiste de una examinación de izquierda a derecha del programa fuente. De esta forma podemos diseñar que las tareas del analizador léxico, del analizador sintáctico, del análisis de contexto y la generación de código intermedio pueden ser realizadas en una sola pasada a como se muestra en la figura.

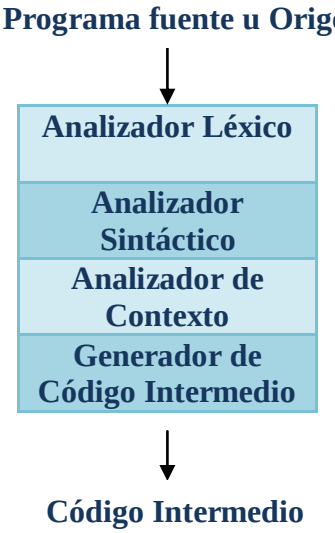


Fig. 1.12. Traductor de un sola pasada.

Nosotros desarrollaremos un compilador que emplea un scanner para reconocer la lista de tokens válidos que conforman el programa fuente e inicializará la tabla de símbolos y atributos. El *scanner* debe contener una función de error para comunicar al usuario de posibles errores. El *parser* tomará la lista de tokens y comprobará (mediante la creación de un árbol de parseo) que la lista de tokens forma una cadena válida para la gramática especificada. A su vez, el parser actualiza con los datos que crea necesarios. La tabla de símbolos y atributos. La fase de análisis de contextos se realiza en conjunto con el parser y para producir la generación de código intermedio se incorporarán en la gramática los símbolos de acción (realizar las asignaciones y operaciones) que son los que se encargaran de ir construyendo las instrucciones de código intermedio necesarios para la traducción final del programa original en el lenguaje destino.

Este procedimiento que desarrollaremos se llama parseo recursivo descendente, el cual utiliza el método de parseo de arriba hacia abajo. Este toma su nombre del hecho que su implementación se hace por medio de rutinas posiblemente recursivas que descienden a través del árbol de parseo a medida que van reconociendo el programa fuente. La idea es que cada procedimiento o función de parseo está asociado con un símbolo no terminal. Para que el parser sepa de antemano que función es la que se aplicará, es que se utiliza el concepto de la variable **lookahead** para seleccionar la alternativa más adecuada del lado derecho de una producción en la gramática. Note que el compilador no construye un árbol de parseo propiamente, pero que este es implícitamente definido por la secuencia de llamadas de los procedimientos sintácticos.

Tema 2. ESQUEMA GENERAL DE UN COMPILADOR

Objetivos

- Identificar las partes que componen la estructura del compilador SLANG AMPLIADO
- Identificar la sintaxis para el lenguaje SLANG AMPLIADO.
- Representar las trazas de un parser recursivo descendente sobre un input.

El proyecto de la asignatura de Diseño de Compiladores será construir un compilador para SLANG AMPLIADO, que es un subconjunto de un lenguaje de programación de alto nivel. El diseño de SLANG AMPLIADO es el resultado de unir dos objetivos.

- Primero, el lenguaje necesita ser lo suficientemente complejo como para ilustrar adecuadamente el proceso de compilación. Por esa razón es que se han incluido los procedimientos.
- Segundo, el lenguaje necesita ser pequeño y simple. Por esto los procedimientos no poseen parámetros y el porqué de un solo tipo de datos, enteros. El lenguaje, sin embargo, está definido de tal forma que resulta fácil adicionar nuevos constructores.

1. El lenguaje provee de expresiones aritméticas que tienen constantes, variables y expresiones entre paréntesis como operandos.
2. El único tipo de datos es el entero. Los enteros contienen sólo dígitos del 0 al 9.
3. Las prioridades de los operadores aritméticos y relaciones son:
   a. Los operadores unarios de negación (NEG) y valor absoluto (ABS) tienen la mayor prioridad.
   b. Los operadores de multiplicación (*), división (/) y modulo (I), (%) porcentaje y (^) la potencia en la siguiente prioridad, después
   c. Los operadores de suma(+) y resta(-), y finalmente
   d. Los operadores relacionales, representados por =, <>, <, <=,>,>=.
4. El vocabulario de SLANG AMPLIADO consiste de letras, dígitos, palabras reservadas y símbolos especiales.
5. Los comentarios son delimitados por llaves { y }, pueden aparecer en cualquier parte del programa y deben ser ignorados por el scanner. No se aceptarán comentarios “corridos”.
6. El lenguaje no será sensitivo a minúsculas o mayúsculas.
7. Los espacios en blanco, tabuladores o fin de línea, sirven para separar los tokens y deben ser ignorados por el compilador.
8. Las cadenas de caracteres comienzan y terminan con comillas dobles y contienen cualquier secuencia de caracteres imprimibles. Si una comilla aparece dentro de la cadena de caracteres, esta debe ser repetida.
9. Los identificadores sirven para nombrar variables o procedimientos. Un identificador inicia siempre con una letra y opcionalmente puede ir seguido de una o más letras o uno o más dígitos. La longitud máxima de un identificador es de 32 caracteres.
10. Cada identificador en la parte de declaración de constantes denota un número.

11. El alcance de una constante es dentro del block en el cual su declaración ocurre.
12. Cada identificador en la parte de declaración de variables denota una variable.
13. El alcance de una variable es dentro del block en el cual su declaración ocurre.
14. Los identificadores en la parte de declaración de procedimientos denotan nombre de procedimiento.
15. El alcance de un procedimiento es dentro del block en el cual su declaración ocurre.
16. Un identificador no puede ser declarado dos veces en el mismo block.
17. La orden de asignación debe ocurrir dentro del alcance del identificador de la parte izquierda, y el identificador debe denotar una variable.
18. La orden de llamada debe ocurrir dentro del alcance del identificador y el identificador debe denotar un procedimiento.
19. La orden de llamada debe ocurrir dentro del alcance del identificador y el identificador debe denotar un procedimiento (CALL nombre_proc).
20. La llamada a un procedimiento debe ocurrir después que el identificador del procedimiento ha sido declarado.

Símbolos Especiales

open_token	:	(
close_token	:	)
list_token	:	,
period_token	:	.
separator_token	:	;
assign_token	:	:=
plus_token	:	+
minus_token	:	-
times_token	:	*
over_token	:	/
pow_token	:	^
porcen_token		%
modulo_token	:	|
equal_token	:	=
not_equal_token	:	<>
less_than_token	:	<
less_or_equal_token	:	<=
greater_than_token	:	>
greater_or_equal_token	:	>=

Palabras reservadas

begin_token	:	BEGIN_
end_token	:	END_
int_token	:	INT_
var_token	:	VAR_
procedure_token	:	PROCEDURE_
call_token	:	CALL_
read_token	:	READ_
cos_token	:	COS_
write_token	:	WRITE_
if_token	:	IF_
then_token	:	THEN_
else_token	:	ELSE_
fi_token	:	FI_
while_token	:	WHILE_
do_token	:	DO_
od_token	:	OD_
negate_token	:	NEG_
absolute_token	:	ABS_
number_token	:	NUMBER_ o DIGITO_
to_token	:	TO_
for_token	:	FOR_

Reglas para otros tokens

Letra = a..z|A..Z
Dígito = 0..9
Identificador = Letra{Letra|Dígito}*
Número = {Digito}$^{+}$

Ahora definiremos la **gramática libre de contexto** (CFG) para el lenguaje SLANG AMPLIADO.

program_declaration → block period_token
block → begin_token declaration_part statement_part end_token
declaration_part → {constant_declaration | variable_declaration}* {procedure_declaration}*
constant_declaration → type_declarer identifier equal_token number {list_token identifier equal_token number}* separator_token
type_declarer → int_token
variable_declaration → var_token type_declarer identifier {list_token identifier}* separator_token
procedure_declaration → procedure_token identifier block separator_token
statement_part → statement separator_token {statement separator_token}*
statement → assign_statement
statement → if_statement
statement → while_statement
statement → call_statement
statement → read_statement
statement → write_statement
assign_statement → identifier assign_token expression
if_statement → if_token relation then_token statement_part [else_token statement_part] fi_token
else_part → [else_token] statement_part
while_statement → while_token relation do_token statement_part od_token
for_statement → for_token relation do_token statement_part od_token
call_statement → call_token identifier
read_statement → read_token open_token identifier {list_token identifier}* close_token
cos_funtion → cos_token open_token identifier close_token

write_statement → write_token open_token expression {list_token expresion}* close_token

expression → [unary_op] term {operator term}*
term → number | identifier| open_token expression close_token | number to_token number
unary_op → negate_token
unary_op → absolute_token
operator → plus_token
operator → minus_token
operator → times_token
operator → over_token
operator → pow_token
operator → porcen_token
operator → mod_token
relation → open_token expression relational_operator expression close_token | expression relational_operator expression
relational_operator → equal_token

relational_operator → not_equal_token
relational_operator → less_than_token
relational_operator → less_or_equal_token
relational_operator → greater_than_token
relational_operator → greater_or_equal_token
identifier → id_token
number → number_token

Ejemplos de programas en SLANG AMPLIADO.

```
BEGIN VAR INT number, sum;
     sum:=0;
     READ(number);
     WHILE(number<>0)DO
              sum:=sum+number;
              READ(number);
    OD;
    WRITE(sum);
END.
```

```
BEGIN VAR INT number;
          PROC reverse
          BEGING
                    WRITE(number | 10);
                     IF(number >= 10)THEN
                              number := number / 10;
                             CALL reverse;
                     FI;
END;

         READ(number);
         WHILE(number <> 0) DO
                  CALL reverse;
                  READ(number);
         OD;
END.
```

```
BEGIN VAR INT number, sum;
     sum:=0;
     READ(number);
     FOR i=0 TO 10  DO
              sum:=sum+number;
              READ(number);
    OD;
    WRITE(sum);
END.
```

Ejemplo: Utilización de varios procedimientos en un programa en SLANG AMPLIADO

```
BEGIN PROC p1
        BEGIN PROC p3
                BEGIN...END;
            PROC p4
                BEGIN VAR INT x;... END;

        END;
        PROC p2
        BEGIN PROC p5
                BEGIN PROC p6
                        BEGIN...; CALL p1; ...END;

                        CALL p6;

                 END;

                 CALL p5;

          END;

          CALL p2;
END.
```

¿Cuál es la estructura de árbol de los procedimientos anidados de este programa?

Construcción de un compilador

Teniendo la definición del lenguaje fuente, SLANG AMPLIADO, estamos en posición de discutir la traducción de programas escritos en SLANG AMPLIADOen un lenguaje ensamblador. Iniciaremos la presentación de un scanner y un parser libre de contexto. La siguiente implementación será el análisis de contexto y para esto se debe tener buen conocimiento del manejo de la Tabla de Símbolos y Atributos.

Scanner

El análisis léxico del programa fuente es conocido como escaneo y es la primera fase del compilador. Nuestro scanner tendrá que realizar las siguientes funciones:

1. Traducir el programa de entrada (cadena de caracteres almacenados en archivo) en una serie de tokens los que serán almacenados en una lista enlazada.
2. Diferenciar las palabras reservadas de los identificadores.
3. Eliminar los caracteres innecesarios como espacios en blancos y tabuladores.
4. Llevar el control del número de líneas del programa fuente.
5. Advertir de errores léxicos encontrados.

Un token es un nombre que se le da a un conjunto de caracteres que tienen un significado especial para el lenguaje de programación.

Todos los identificadores son implícitamente declarados y no más largos que 32 caracteres. Los Identificadores deben comenzar con una letra y son compuestos por letras, y dígitos.

Nuestro scanner será una función sin argumentos que devuelve tokens.
Tokens scanner (void);

Si se desea se puede implementar un tipo de dato enumerado llamado boolean de la siguiente manera:

Typef enum boolean_type{
TRUE = 1, FALSE = 0 } boolean;

Definiremos un tipo de dato enumerado llamado tokens que representara el conjunto de tokens de SLANG AMPLIADO.

Typef enum token_type{
OPEN_, CLOSE_, LIST_, PERIOD_, SEPARATOR_, ASSING_, PLUSOP_, MINUSOP_, TIMESOP_, OVEROP_, MODOP_, EQUAL_, NOTEQUAL_, LESS_, LESS_EQ_, GREAT_, GREAT_EQ_, BEGIN_, END_, ID_, INTEGER_, NUMBER_, VAR_, PROCEDURE_, CALL_, READ_, WRITE_, IF_, THEN_, ELSE_, FI_, WHILE_, DO_, OD_, NEGATE_, ABS_ } tokens;

Para hacer uso de los nombres de los tokens en el contexto de las líneas de código, incluimos el siguiente código:

Char *tokenName[]= {
"OPEN_", "CLOSE_", "LIST_", "PERIOD_", "SEPARATOR_", "ASSING_", "PLUSOP_", "MINUSOP_", "TIMESOP_", "OVEROP_", "MODOP_", "EQUAL_", "NOTEQUAL_", "LESS_", "LESS_EQ_", "GREAT_", "GREAT_EQ_", "BEGIN_", "END_", "ID_", "INTEGER_", "NUMBER_", "VAR_", "PROCEDURE_", "CALL_", "READ_", "WRITE_", "IF_", "THEN_", "ELSE_", "FI_", "WHILE_", "DO_", "OD_", "NEGATE_", "ABS_"};

Se requiere mucho cuidado cuando se efectúa el scaneo. En particular, es necesario llevar un control sobre el comienzo del próximo token en orden de reconocer el final del token actual. Para hacer esto, el scanner hace la inspección del siguiente carácter del input. Si el carácter no es legal para iniciar un nuevo token, un error es encontrado. Un mensaje de error debe ser impreso y después debemos intentar una recuperación del error. Una manera simple de hacer esto es saltar el carácter ilegal y recomenzar el scaneo. Este proceso continua hasta que el comienzo de un token es encontrado. En conclusión siempre se hace el chequeo de que la secuencia de caracteres posibles mas larga forme un token legal.

Para SLANG AMPLIADO, todo lo que necesitamos es una variable tipo carácter que llamaremos **lookahead**, la que puede ser obtenida de la lectura del código fuente utilizando la función fgetc(ptrToFile). Este carácter puede ser convenientemente regresado, según sea el caso, al input original usando la función **ungetc (char, prtToFile).**

A continuación mostramos parte del código del loop principal de un scanner que puede reconocer identificadores y literales enteras en SLANG AMPLIADO. Este también salta espacios en blanco. Por simplicidad, asumimos un final de línea en cada línea. En C, este es usualmente llamado "nueva línea" y es denotado por la secuencia de escape **'\n'.**

Scan.h

```
char        buffer[32];
FILE        *ptrToFile;
int         numLin=0;
tokens      resptemp;

tokens scanner(void)
{
   char   input,c;
   int    i;


while((input=fgetc(ptrToFile)) !=EOF){
        limpiar_buffer();
```

```
        if(input== '\n')
            numLin++;
        if(isspace(input))
            continue;
        else if(isalpha(input) {
                buffer[i ++]= input;
        for (c == fgetc(ptrToFile); isalnum(c); c = fgetc(prtToFile))
                buffer[i++]=c;
        ungetc(c, ptrToFile);
        resptem=chequear_reservada();
        return(resptemp);
    }
else if(isdigit(input)){
        buffer[i++]=input
        while(isdigit((c=fgetc(ptrToFile))))
                buffer[i++]=c;
        ungetc(c, stdin);
        resptemp = NUMBER_;
        return(resptemp);
    }
else...
  ...
  ...
  ...
else lexical_error(numLin,input);
}/*while*/
}/*scanner*/
```

Otras funciones que se deberán implementar para completar el scanner:

- void limpiar_buffer(void);
- token chequear_reservada(void);
- void lexical_error(token t);

Observación:

No perder nunca la cabecera de la lista de tokens

Por otro lado la forma inicial del programa compilador será:

Compilador.c

```
#include<stdio.h>
...
...
#include "listT.h"
#include "scan.h"
...

void main()
{
     char     listar, nombre[12];
     tokens   t;
     listTokens *cabLT=NULL;

    system("cls");
    printf("Nombre del archivo fuente: ");
    gets(nombre);
    if(ptrToFile=fopen(nombre, "r"))==NULL)
    {
       printf("\n El archivo no se pudo abrir ");
       exit(0);
     }
     else if(!ferror(ptrToFile)&& !feof(ptrToFile)){
               t=scanner();
               insertarLT(cabLT, t);
  }
  printf("\nFin del scaneo ");
  printf("\n¿Desea visualizar la lista de tokens(S/N)?");
  gets(listar);
  if(listar='s') || (listar='S'))
     visualizarLT(cabLT);
  /*Aqui se llamará al parser*/
   fclose(ptrToFile);
 }
```

El archivo **listaT.h** definirá una estructura de tipo lista de la siguiente manera:

```
 typedef struct lt{
        tokens       t;
        struct lt     *sig;
}listaTokens
```

También se deben incluir las funciones:

- insertarLT(listaTokens *cab, tokens t);
- visualizarLT(listaTokens *cab);
- borrarLT(listaTokens *cab);

Parser

Primero, repetiremos un poco los principios del parseo recursivo descendente dejando ver lo fácil que es extenderlo para nuestro lenguaje SLANG AMPLIADO.

La idea básica de un parseo recursivo descendente es que cada procedimiento de parseo es asociado a un símbolo no terminal de la gramática. El parseo se comienza llamando el procedimiento asociado con el símbolo de inicio. Durante el parseo, cada terminal en el lado derecho de una producción es verificado con el símbolo token_actual (el cual es el símbolo al cual se esta apuntando actualmente en la lista de tokens que produjo el scanner) y cada no terminal resultará en el llamado a su procedimiento asociado. De esta forma, los procedimientos de parseo descienden a través del árbol de parseo a medida que se reconoce la lista de token del programa fuente. El árbol de parseo es definido explícitamente por las secuencia de llamadas de los procedimientos sintácticos o de parseo. En la práctica, el árbol de parseo puede ser también generado por el compilador.

Si el lado derecho de una producción consiste de un número de alternativas (al menos 2), entonces el símbolo token_ahead determinará cual de las alternativas debe ser seleccionada. Si todas las alternativas comienzan con un token diferente, y el token_ahead coincide con uno de ellos, entonces la alternativa iniciando con este token debe ser la seleccionada para llamar a su procedimiento de parseo asociado.

Ejemplos:

add_operator	→	plus_token | minus_token
multiply_operator	→	times_token | over_token | modulo_token
unary_operator	→	negate_token | absolute_token

Todas estas producciones consisten de un token al lado derecho y esto se traduciría al siguiente código:

```
AddOperator(void)
{
     tokens  tempToken;

    tempToken= token_actual->nombre;

    if(tempToken == plus_token) then
          match(plus_token);
    if(tempToken == minus_token) then
      match(minus_token);
}
```

Al ser llamado el parser, la variable token_actual, del tipo listaTokens, deberá estar apuntando al inicio o cabecera de la lista de tokens que produjo el scanner. Es importante no perder nunca la cabecera de la lista de tokens.

La función match (tokens t) realiza dos funciones:

- Verifica si el token t, pasado como argumento de esta función, coincide con el nombre del puntero token_actual.
- Si la comparación es exitosa, se avanza el puntero del token_actual en la lista al elemento siguiente de la lista de tokens y regresa el control de ejecución.
- Si t es diferente del token_actual, entonces se produce un llamado a la función syntax_error(tokens t, int numLin). Recordar que el parser debe tratar de recuperarse de este error y seguir el proceso después de haber emitido el error correspondiente.

```
void match(token t)
{
    tokens tempToken;

   tempToken = get_token();

   if(temp == t)
  {
        token_actual = token_actual-> siguiente;
        return;
   }
 else
  {
       syntax_error(tempToken, linea);
  }
}
```

También puede servir de ayuda la función **get_token(void)**, la que haciendo uso de una variable temporal tipo tokens obtiene el nombre del token del puntero token_actual de la lista de tokens. Es importante que se verifique que esta operación no afecte al puntero token_actual en la lista de tokens.

```
tokens get_token(void)
{
     tokens temp;

     temp = token_actual->nombre;
     return(temp);
}
```

A continuación mostramos algunos procedimientos de parseo que formarían el programa parser para nuestra lenguaje SLANG AMPLIADO.

Parser.h

```
/* Declaraciones de constantes y de tipos necesarios*/

   listaTokens token_actual;

void program_declaration(void)
{
      token_actual = cabLT;
      block();
      match(PERIOD_);
}

void block(void)
{
      match(BEGIN_ );
     declaration_part();
     statement_part();
     match(END_);
}

 void declaration_part(void)
{
     int control = 0;
     tokens tempToken = get_token();


     while((tempToken == INT_) || (tempToken == VAR_))
     {
            if(tempToken == INT_)
                  constant_declaration();
           else if(tempToken == VAR_)
                 variable_declaration();
           control = 1;
   }
   if(control == 1)
    {
         tempToken = get token();
         while(tempToken == PROCEDURE_)
         {
              procedure_declaration();
         }
     }
     else if(control == 0)
     {
            while(tempToken == PROCEDURE_)
            {
                 procedure_declaration();
            }
```

```
    }
}

void constant_declaration(void)
{
    tokens tempToken;

    type_declarer();
    identifier();
    match(EQUAL_);
    number();
    tempToken = get_token();
    while(tempToken == LIST_)
    {
        match(LIST_)
        identifier();
        match(EQUAL_);
        number();
    }
    match(SEPARATOR_);
}

void type_declarer(void)
{
    match(INT_);
}

void variable_declaration(void)
{
    tokens tempToken;

    match(VAR_);
    type_declarer();
    identifier();
    tempToken = get_token();
    while(tempToken == LIST_)
    {
        match(LIST_);
        identifier();
    }
    match(SEPARATOR_);
}

void procedure_declaration(void)
{
    match(PROCEDURE_);
    identifier();
    block();
    match(SEPARATOR_);
}
```

TAREA: Se debe continuar con las demás funciones hasta completar todas las reglas de la gramática.

Ahora el archivo **Compilador.c** se modifica para incluir la llamada al parser.

Compilador.c

```
#include<stdio.h>
...
...
#include "listT.h"
#include "scan.h"
...

void main()
{
    char        listar, nombre[12];
    tokens      t;
    listaTokens  *cabLT=NULL;

    system("cls");

    printf("Nombre del archivo fuente:");
    gets(nombre);
    if(ptrToFile=fopen(nombre,"r"))==NULL)
   {
      printf("\nEl archivo no se pudo abrir");
      exit(0);
   }
   else if(!ferror(ptrToFile) && !feof(ptrToFile)){
            t=scanner();
            insertarLT(cabLT,t);
 }
 printf("\nFin del scaneo");
 printf("\n¿Desea visualizar la lista de tokens(S/N)?");
 gets(listar);
 if((listar='s') || (listar='S'))
     visualizarLT(cabLT);
 program_declaration();
 printf("\nFin de parseo");
 fclose(ptrToFile);
}
```

Manejo de la Tabla de Símbolos y Atributos (TSA)

La definición del lenguaje SLANG AMPLIADO requiere que cada ocurrencia aplicada de un identificador sea únicamente asociada con una ocurrencia de definición. La ocurrencia de definición (declaración) determina el alcance (del bloque), la clase (constante, variable o procedimiento) y el tipo (entero) del identificador.

La asociación única entre las ocurrencias aplicadas y las de definición implica que debe de existir solamente una ocurrencia de definición del identificador en el mismo bloque.

Cada ocurrencia aplicada de un identificador debe de estar en el alcance de su definición de ocurrencia de la clase y el tipo correcto.

La información almacenada de un identificador forma lo que se llama el record semántico. Los record semánticos se mantienen en lo que se llama la Tabla de Símbolos y Atributos. La Tabla de Símbolos y Atributos debe de proporcionar un acceso rápido a la información recolectada acerca de las ocurrencias de definición de todos los identificadores.

Diseño de la Tabla de Símbolos y Atributos

Primero enfatizaremos sobre las reglas del alcance y visibilidad de los identificadores en SLANG. Un identificador esta delimitado a un objeto en una declaración. El alcance de la delimitación es el bloque del procedimiento en el cual la declaración ocurre. Una delimitación que se mantiene en un bloque de un procedimiento también se mantiene en los bloques de sus procedimientos subordinados, al menos que el identificador sea redefinido en un bloque subordinado. De todo esto podemos derivar las siguientes reglas de visibilidad de los identificadores:

- En cualquier punto del programa, solamente los identificadores declarados en el bloque actual y en los bloques subordinados son visibles.
- Si un identificador es declarado en más de un bloque, entonces solamente la declaración más cercana al punto de la ocurrencia es visible.

Ejemplo 2.1. Considere el programa siguiente:

```
BEGIN                                                    1
    VAR INT p,w,g;                                       2
    p:=0;                                                3
   w:=0;
   g:=30;                                                4
  PROC d;                                                5
        BEGIN                                            6
              VAR INT a,r,w;                             7
              PROC z                                     8
                    BEGIN                                9
                          VAR INT e,p,w;                 10
```

```
                                    w:=10;                          11
                                    e:=25;                          12
                                    p:=w+e+g;                       13
                                END;                                14
                            w:=20;                                  15
                           r:=50;                                   16
                           a:=p+r+w;                                17
                    END;                                            18
            WRITE(p+w);                                             19
END.                                                                20
```

Las declaraciones visibles en diferentes puntos del programa se muestran en la siguiente tabla:

Línea	Declaraciones accesibles
11	a(2), d(1), e(3), p(3), r(2), w(3), z(2), g(1)
18	a(2), d(1),p(1), r(2), w(2), z(2), g(1)
19	d(1),p(1), w(1), g(1)

Fig. 2.1. Tabla de declaraciones visibles.

Usualmente la Tabla de Símbolos y Atributos se implementa como una tabla hash o como un árbol binario. Para lenguajes estructurados por bloques, como SLANG AMPLIADO, existe una opción de tablas individuales para cada alcance o una tabla global. Nosotros implementaremos una pila de listas, donde cada nuevo nodo de la pila indica el avance del alcance y este a su vez posee un puntero a una lista de records semánticos correspondientes a las ocurrencias de los identificadores dentro del bloque actual.

Un record semántico consta de los siguientes campos:

- El nombre del identificador.
- La clase.
- El tipo.
- El valor (para variables y constantes)
- Un puntero al próximo record semántico.

La lista de la definición de records semánticos es organizada como lista push-down. Cuando se este parseando la ocurrencia de la definición de un identificador debe ser creada e insertada en el frente de la lista de definición de records. Una definición de record será removida de la lista cuando salimos del bloque en el cual la declaración ocurrió. Este esquema garantiza que, en cualquier tiempo, solamente las declaraciones del bloque actual y de los subordinados son accesibles y que la declaración más cercana al punto de compilación es encontrado primero.

Cuando salimos de un bloque, todas las definiciones de records durante el parseo de este bloque deben ser removidas de la lista de definiciones.

Todo lo antes expuesto se traduce a definir los siguientes tipos de datos para ser utilizados por los records semánticos:

```
#define MAX_ID_LEN              33
typedef char string [MAX_ID_LEN];

typedef enum {const_kind, var_kind, proc_kind, unknown_kind} Clase;

typedef enum {int_type, no_type, unknown_type} Tipo;

typedef struct record {
        string      idNombre;
        Clase       idClass;
        Tipo        idTipo;
        Int         idValor;
        struct record    *sig;
} RecordSemantico;
```

Al inicio del proceso de parseo, la Tabla de Símbolos y Atributos deberá de ser inicializada y se inicializa el nivel de alcance a 1.

Cuando entramos en un bloque, entramos en nuevo alcance; por lo tanto, se debe crear a otro nodo de la pila para almacenar en la lista correspondiente los record que sean necesarios. Esta operación es realizada por el procedimiento push(&TSA), la cual incrementa el nivel del alcance.

Cuando salimos de un bloque, el alcance actual se elimina y todos los identificadores definidos en la lista del nivel actual se eliminan y el puntero de la Tabla de Símbolos y Atributos baja un nivel en la pila. Estas acciones son realizadas por el procedimiento pop(&TSA), la cual decrementa el nivel del alcance.

El nivel del alcance del bloque que está siendo actualmente parseado se va almacenando en la variable global **nivelglobal.**
La función

RecordSemantico *definir_record(void);

Deberá asignar dinámicamente memoria para un objeto del tipo RecordSemantico.

La siguiente tarea es analizar la gramática del lenguaje SLANG AMPLIADO y decidir en que puntos del parseo se deberán incluir las llamadas a los que se denominaran **rutinas semánticas.**

Traduciendo SLANG AMPLIADO

Comenzaremos a trabajar en la real traducción de SLANG AMPLIADO. Primero, debemos decidir para que máquina generaremos código y en que forma (ensamblador u otros) el código generado será producido. Por simplicidad, usaremos códigos ensamblador para máquinas de tres direcciones. Instrucciones para este tipo de máquinas tienen la forma

OP A, B, C

en el cual OP es un op-code (o pseudo-op), A y B designan operandos de alguna especificada operación, y C especifica la localización donde le resultado de la operación será almacenado. Los operadores pueden ser nombres de variables o literales. Para algunos OP's, A o B o C puede no ser usada (por ejemplo, en una instrucción halt). El formato de nuestro output será una cadena de caracteres.

El código destino generado de una máquina virtual simple, este puede ser usado para manejar un interprete, o las instrucciones mismas pueden ser expandidas por un generador de código más sofisticado en código para máquinas reales. De hecho, nuestro código generado es muy similar a cuádruples, una manera común usada en la representación intermedia. Este punto ilustra una interesante propiedad del conjunto de instrucciones virtuales. Este conjunto puede ser visto como una representación intermedia o como la salida de un compilador, dependiendo de como estas instrucciones son utilizadas.

Temporeros

Durante la compilación, es frecuentemente necesario utilizarlo temporalmente espacio para asignar celdas de almacenamiento, conocidas como ***temporeros***, para guardar resultados intermedios de una computación. Para nuestro compilador de SLANG AMPLIADO hemos pensado en los temporeros como variables internas que son implícitamente declaradas cuando son necesitadas. Esta técnica trabaja bien en SLANG AMPLIADO puesto que todas las variables son implícitamente declaradas. En los compiladores para lenguajes más realísticos, los temporeros son utilizados para guardar valores de registros, siendo utilizados cuando no hay registro disponible (la tabla de registro es llena). Usaremos la convención en la cual las variables internas usadas como temporeras son de la forma Temp&N, donde N es el índice de la temporera, comenzando en 1. Puesto que & no puede aparecer en una variable ordinaria de SLANG AMPLIADO, no habrá ningún conflicto.

Símbolos de Acción

Como se explicó anteriormente, el trabajo fuerte de la traducción es realizado por las Rutinas Semánticas llamadas por el parser. Los Símbolos de Acción pueden ser agregados a la gramática para especificar cuando en el procedimiento semántico debe tomar lugar. Los símbolos de acción son denotados por **#nombre** y puede ser puesto en el lado derecho de la producción. Correspondiente a cada símbolo de acción existe una rutina semántica. Así el símbolo **#add_op** corresponde a la rutina semántica **add_op().**

Los símbolos de acción no tienen impacto en el lenguaje reconocido por un parser derivado por una CFG. Así, ellos no son realmente parte de la sintaxis de la CFG especificada. En este contexto, los símbolos de acción sirven como comentarios de la CFG, indicando cuando alguna acción semántica necesita ser ejecutada.

Información Semántica

Un tópico importante en el diseño de las rutinas semánticas es la especificación de los datos sobre los cuales ellas operarán y la información que ellos producen. Nuestra solución será asociar un **record semántico** con cada clase de símbolo gramática. Cada símbolo tendrá un record distinto, conteniendo la información adecuada para este símbolo.

El récord semántico para un no terminal es creado por una rutina semántica que tiene acceso a la información acerca de los símbolos en el lado derecho de la producción. Si algunos de los símbolos son no terminales, sus correspondientes records semánticos vienen de las llamadas a las rutinas semánticas especificadas dentro de sus propias producciones. Para ver como esto trabaja, considere
expresión→ term{operator term #gen_infix}*

En la TSA debe de existir un récord semántico para cada uno de los term en el lado derecho de la producción. Estos récords semánticos almacenan datos acerca de cada uno de los operandos (por ejemplo, donde está almacenado o cual es su valor). Cuando **operator()** es llamada, esta debe dar esos récords como parámetros. Estos son usados para generar el código apropiado, después actualizan el récord semántico correspondiente al identificador de la expresión (del lado izquierdo) que almacenará información necesaria acerca de la expresión justamente procesada.

```
/*para operadores*/
typedef struct operator{
      enum op{PLUS,MINUS,TIMES,OVER,MOD}operator;
}op_rec;

/*Tipos de expresiones*/
  enum expr{IDEXPR,CONSTANTEXPR,TEMPEXP};

/*para expresión*/
typedef struct expression
{
    enum  expr kind; //Tipo;
    struct
   {
       string name;        /*for IDEXPR, TEMPEXP*/
       int val;            /*for CONSTANTEXPR*/
    };
}expr_rec;
```

Utilizaremos algunas rutinas auxiliares en nuestro compilador:

Parser.h

```
pila *TSA=NULL;

void program_declaration(void)
{
    tokens *token_actual=cabLT;
    push(&TSA);
    block();
    match(PERIOD_);
    Liberar_memoria(&TSA);
}

void block(void)
{
    match(BEGIN_);
    declaration_part();
    statement_part();
    match(END_);
}

void declaration_part(void)
{
     int control=0;
     tokens tempToken=get_token();

     while((tempToken==INT_) || (tempToken==VAR_))
     {
         if(tempToken==INT_)
                 constant_declaration();
         else if(tempToken==VAR_)
                variable_declaration();
        control=1;
    }
    if(control==1)
     {
         tempToken=get_token();
         while(tempToken==PROCEDURE_)
         {
              procedure_declaration();
         }
     }

       else if(control==0)
  {
      while(tempToken==PROCEDURE_)
      {
```

```
            procedure_declaration();
          }
        }
}

void constant_declaration(void)
{
       tokens tempToken;
       RecordSemantico *rs_cd=nuevo_record();

       rs_cd->idClase=const_kind;
       type_declarer(rs);
       identifier(rs);
      Agregar_elemento_lista(&TSA->elem,rs)
      match(EQUAL_);
      number(rs);
      tempToken=get_token();
      while(tempToken==LIST_)
     {
          rs_cd=Nuevo record();
          rs_cd->idClase=const_kind;
          rs_cd->idTipo=int_type;
          match(LIST_);
          identifier(rs);
         Agregar_elemento_lista(&TSA->elem,rs);
         match(EQUAL_);
         number(rs);
      }
        match(SEPARATOR_);
}

void type_declarer(RecordSemantico *rs)
{
       rs->idTipo=int_type;
       match(INT_);
}

void variable_declaration(void)
{
       tokens tempToken;
      RecordSemantico  rs_vd=nuevo_record();

      match(VAR_);
      rs_vd->idClass=var_kind;
      type_declarer(rs);
      identifier(rs);
     Agregar_elemento_lista(&TSA->elem,rs);
     tempToken=get_token();
     while(tempToken==LIST_)
    {
```

```
        rs_vd=Nuevo_record();
        rs_dv->idClase=var_kind;
        rs_dv->idTipo=int_type);
        match(LIST_);
        identifier(rs);
       Agregar_elemento_lista(&TSA->elem, rs);
   }
    match(SEPARATOR_);
}

void procedure_declaration(void)
{
      RecordSemantico     rs_pd=nuevo_record();

      match(PROCEDURE);
      rs_pd->idClase=proc_kind;
      identifier(rs);
      push(&TSA);
      block();
      match(SEPARATOR_);
}
```

Símbolos de Acción

Los símbolos de acción son una notación de llamadas a funciones que procesaran la información necesaria en la TSA para que la traducción real pueda darse de acuerdo a lo que el programador especifica en su programa fuente.
Una producción con símbolo de acción agregado luce como:
<ident> → ID_ #process_id

Esta producción es de mucha ayuda ya que ID_ aparece en diferentes contextos en la gramática de SLANG AMPLIADO, y se necesita una llamada a **process_id()** inmediatamente después de que el parser verifique la ocurrencia de un identificador (para acceder a los caracteres en el **token_buffer** y construir un récord semántico apropiado).

También para la generación de código utilizaremos una serie de funciones y procedimientos auxiliares para escribir en un fichero las instrucciones en ensamblador que vamos interpretando del lenguaje fuente a medida que se efectúa el parseo. A continuación detalles de algunas de estas funciones. No limita a que el usuario defina sus propias funciones si lo cree conveniente o encuentra una forma más fácil de realizar la misma implementación.

generate() tomara cuatro argumentos tipo string correspondiente al código de la operación, dos operandos, y el campo del resultado. Este producirá una instrucción correctamente formateada en un archivo de salida.

extract() tomara un record semántico y devolverá una cadena correspondiente a la información semántica que contiene. La cadena puede ser un identificador, un op-code,

un número, y así. La información extraída sirve de input para **generate()** para crear la instrucción completa.

Debido a que estaremos generando instrucciones de lenguaje ensamblador que permitirán al ensamblador asignar almacenamiento para variables, no necesitamos almacenar ninguna información sobre dirección como un atributo de un identificador. De hecho, atributos explícitos no serán utilizados. La única información de interés acerca del identificador es cuando este ya existe en la Tabla de Símbolos, así el compilador conocerá si puede generar una instrucción que causará la asignación del espacio requerido. Una rutina adicional que necesitaremos para nuestra Tabla de Símbolos (TSA) es:

```
/* ¿Esta s en la Tabla de Símbolos?*/
int Buscar_TSA(string s);
```

Buscar_TSA() chequeará cuando una entrada (record semántico) llamada ***s*** está en la tabla de símbolos, la búsqueda iniciará a partir del nivel de alcance actual y debe detenerse en la primera ocurrencia del nombres ***s***.

Ya que la única información de interés acerca un identificador es solamente cuando este existe o no de previo a su declaración en la tabla de símbolos, así el compilador sabrá si este genera una instrucción que causará un problema de asignación. Por lo tanto además de la función Buscar_TSA, definiremos la función **Poner_TSA()**, la cual introducirá incondicionalmente s en la tabla de símbolos.

```
void check_id(string s){

if(!Buscar _TSA(s)){
     Poner_TSA(s);
     Generate("Declare",s,"Integer","");
    }
}
```

Check_id() declarara una variable introduciéndola en la tabla de símbolos y luego generara un directivo ensamblador para reservar espacio para el símbolo. En nuestro lenguaje ensamblador, *Declare* es una instrucción de pseudo-código que declara un nombre al ensamblador y define su tipo. Esto trabaja para variables globales simples no estructuradas.

Otra función que utilizaremos y que podemos implementar es:

```
extern void Enter_TSA(string s);
```

que sirve para poner incondicionalmente el símbolo **s** en la TSA.

```
void Buscar_TSA(string s){

  if(!Buscar _TSA(s)){
     Enter_TSA(s);
     generate("Declare",s,"Integer","");
    }
}
```

Buscar_TSA() generará un directivo ensamblador para reservar espacio para el símbolo. En nuestro lenguaje ensamblador, *Declare* es una instrucción de pseudo-código que declarara un nombre al ensamblador y define su tipo. Esto trabaja para variables globales simples no estructuradas. El ensamblador decide cuanto espacio es requerido por la variable y exactamente donde esta será asignada.

También necesitaremos una rutina para asignar temporeros. A como mencionamos antes, asignaremos temporeros de la misma forma que las variables son asignadas. La única diferencia es que los nombres temporeros serán generados por el compilador y sin ningún significado en un programa en SLANG AMPLIADO. Se utilizarán nombres como Temp&1, Temp&2 y así sucesivamente. La función **get_temp()** asigna registros:

```
char *get_temp(void){
        /*max temporeros asignados hasta ahora*/
        static int max_temp=0;
        static char tempname[MAX_ID_LEN];
        max_temp++;
        printf(tempname,"Temp&%d",max_temp);
        Buscar_TSA(tempname);
        return tempname;
}
```

A continuación tenemos las rutinas auxiliares necesarias que definen las Rutinas Semánticas correspondientes a los símbolos de acción de SLANG AMPLIADO.

```
void start(void){
       /*Inicialización Semántica*/
       push(&TSA);
}

void fin(void){
       /*Generar el código para finalizar el programa*/
         generate("Halt","","","");
}

void assign(expr_rec destino,expr_rec source){
       /*Generate code for assignment*/
       generate("Store",extract(origen),destino.nombre,"");
}

  op_rec process_op(void){
        /*Produce un descriptor de operador*/
        op_rec o;
        if(current_token==PLUSOP_)
                o.operator=PLUS;
        if(current_token==MINUSOP_)
                o.operator=MINUS;
        if(current_token==TIMESOP_)
                o.operator=TIMES;
       if(current_token==OVEROP_)
```

```
                o.operator=OVER;
        if(current_token==MODOP_)
                o.operator=MOD;
        return o;
}

expr_rec gen_infix(expr_rec e1,op_rec op, expr_rec e2){
        expr_rec e_rec;

       /* Un exp_rec con cada temp*/
       e_rec.kind=TEMPEXPR;

      /*Generar código para la operación
         Tomar el resultado temp y preparar
         un récord semántico para el resultado*/
     strcpy(e_rec.name, get_temp());
     generate(extract(op),extract(e1),.extract(e2),e_rec.name);
     return e_rec;
}

void read_id(expr_rec in_var){
     /*Generate code for read*/
      generate("Read",in_var.name,"Integer","");
}

expr_rec process_id(void){
      expr_rec t;

      /*Declarar ID y construir el récord semántico correspondiente*/
      Buscar_TSA(token_buffer);
      t.kind=IDEXPR;
      strcpy(t.name,token_buffer);
      return t;
}

expr_rec process_literal(void){
     expr_rect t;
     /*Convertir literal a representación numérica
        y construir récord semántico*/
      t.kind=LITERALEXPR;
      (void)scanf(token_buffer,"%d",&t.val);
       return t;
}

void write_exp(expr_rec out_expr){
        generate("Write",extract(out_expr),"Integer","");
}
```

Dadas estas rutinas semánticas, ahora podemos volver a nuestras rutinas de parseo para ver de que forma ellas son alteradas para poder manipular la inclusión de los símbolos de acción o rutinas de procesamiento semántico.

Por ejemplo: El procedimiento **expresión()** se altera de la siguiente manera: producirá un dato tipo **expr_rec** como parámetro de salida. Al ser retornada, este **expr_rec** contiene la información semántica acerca de la expresión reconocida por **expresión().**

El cuerpo del procedimiento incluye variable internas utilizadas para almacenar el récord semántico generado por las llamadas de otros procedimientos de parseo y utilizadas en una llamada a la rutina de generación de código **gen_infix().**

```
void expressión(expr_rec *result)
{
    expr_rec left_operand, right_operand;
    op_rec op;

    primary(&left_operand);
    while(next_token()==PLUSOP_ || next_token()==MINUS OP_){
        add_op(&op);
        primary(&rigth_operand);
        left_operand = gen_infix(left_operand,op,right_operand);
     }
     *result = left_operand;
}
```

Ejemplo de un parseo recursivo descendente y traducción

Como ejemplo considere la compilación del siguiente programa en SLANG AMPLIADO:

```
BEGIN VAR INT number,sum;
          sum:=0;
         READ(number);
         WHILE(number<>0)DO
                  sum:=sum+number;
                  READ(number);
          OD;
          WRITE(sum);
END.
```

Tema 3: SCANNER: TEORÍA Y PRÁCTICA

Objetivos

- Define que es expresión regular
  Clasifica los tipos de autómatas y su forma de representar.
- Traduce expresiones regulares a Autómatas finitos
- Identifica la gramática, operaciones y traducción del generador de scanners: FLEX

Introducción

La función primaria del scanner es agrupar los caracteres en tokens. Por esto, el scanner es conocido como el **analizador léxico.**

Una forma sencilla de crear un analizador léxico consiste en la construcción de un diagrama que ilustra la estructura de los componentes léxicos del lenguaje fuente, y después hacer la traducción "a mano" del diagrama a un programa para encontrar los componentes léxicos. Las técnicas utilizadas para construir analizadores léxicos también se pueden aplicar a otras áreas, como, por ejemplo, a lenguajes de consulta y sistema de recuperación de información.

Existen dos razones por las cuales en nuestro estudio hemos separado el analizador léxico y el analizador sintáctico:

1. *Eficiencia:* el procesamiento carácter por carácter consume bastante tiempo y puede ser eficientemente optimizado si lo tratamos localizadamente. Además, el analizador léxico puede identificar y remover construcciones como los comentarios, espacios en blanco, tabuladores, fin de líneas que no juegan ningún papel importante en la compilación.
2. *Diseño:* el analizador léxico y sintáctico trabajan en diferentes niveles de abstracción del lenguaje, el analizador léxico procesa caracteres y palabras mientras que el analizador sintáctico procesa palabras y oraciones. Al separar las especificaciones de las sintaxis y las estructuras de las oraciones, las especificaciones del compilador resultan más leíbles, portables y mantenibles.

Ahora estudiaremos temas relacionados con la creación de scanners para lenguaje de programación comprensivos. El tema aquí es que la estructura de un token puede ser más detallada de los que podemos esperar. Debido a que en todo lenguaje encontramos el concepto de token, es necesario tener una definición precisa de lo que es un token y esto con el objetivo de asegurar que las reglas léxicas son propiamente aplicadas. Las definiciones formales permiten a los diseñadores anticipar el flujo del diseño. Todos los scanners, independientemente de los tokens a ser reconocido, realizan la misma función. Así, escribir un scanner desde cero, significa reimplementar componentes comunes para todos los scanners y esto es una *duplicación de esfuerzos.*

La meta de un generador de scanners es minimizar los esfuerzos en la construcción de un scanner para especificar que tokens el scanner reconocerá. Programar un generador de scanners es un ejemplo de programación no procedimental. Esto es, a diferencia de la programación ordinaria, la cual es llamada procedimental, no le decimos scanner como scanear sino que simplemente le decimos que necesitamos scanear. En UNIX, existen dos generadores conocidos, ScanGen y Lex.

Expresiones Regulares

Las expresiones regulares son un medio conveniente de especificar cierto conjunto de caracteres. Usos:

*Especificar la estructura de los tokens usados en un lenguaje de programación.
*Programa un generador de scanners.

Los conjuntos de cadenas definidas por expresiones regulares son denominados conjuntos regulares. Iniciamos con un conjunto finito de caracteres, o vocabulario, denotado V. este vocabulario es el conjunto de caracteres usados para formar tokens. Una cadena vacía es denotada por. Cadenas nuevas son construidas en V por concatenación. La cadena vacía concatenada con cualquier otra cadena s, resulta en s. Así, sλ ≡ λ hs ≡ s.

Concatenación

Sean P y Q dos conjuntos de cadenas, entonces, la cadena s ∈ (PQ) si y solo si **s** puede ser expresada como $s = s_1 s_2$ tal que $s_1 \in P$ y $s_2 \in Q$.

Los caracteres (,),´,+,-,*, y / se llaman meta caracteres (puntuación y operadores de expresiones regulares). Metacaracteres pueden aparecer entre comillas cuando son usados como caracteres ordinarios, para evitar ambigüedades. Por ejemplo:

Delimitador = **"(" |")" | : | := | ; | , | .. | . |"+"| "–" | "*" | "/" | EOF**

Alternación

Sean P y Q dos conjuntos de cadenas, entonces, la cadena s ∈ (P|Q) si y solo si **s** ∈ P o **s** ∈ Q.

Clausura de Kleene

Sea P un conjunto de cadenas. s ∈ P* si y solo si **s** puede ser definida en cero o mas piezas: $s = s_1 s_2 s_3 \ldots s_n$ tal que cada $s_i \in P$. Una cadena en P* es la concatenación de cero o mas selecciones (posiblemente repetidas) de P.

Expresiones Regulares

Cada expresión regular denota un conjunto de caracteres (un conjunto regular).

- Ø es una expresión regular que denota conjunto vació.
- λ es una expresión regular que denota el conjunto conteniendo solamente a la cadena vacía.
- Una cadena **s** es una expresión regular denotando el conjunto conteniendo solo a **s.**
- Si A y B son expresiones regulares, entonces A|B, AB y A* son también expresiones regulares, denotando alternación, concatenación y la clausura de Kleene del correspondiente conjunto regular.

Notaciones

* P+ denota todas las cadenas consistentes de una o más cadenas de P concatenadas a la vez.
  $P^* = (P^+ \mid \lambda)$ y $P^+ = PP^*$
* Si A es un conjunto de caracteres, Not(A) = (V – A) denota todos los caracteres en V no incluidos en A.
* Si **S** es un conjunto de cadenas, Not(S) = (V* - S).
* Si k es una constante, el conjunto A^k representa todas las cadenas formadas concatenando k veces la cadena A.

Ejemplo: ahora ilustraremos como expresiones regulares pueden ser usadas para especificar tokens.

Sean D = (0 | 1 | 2 | … | 9) y L = (A | B | … | Z), entonces:

* Un comentario que comienza con -- y termina con el fin de línea (eol) puede ser definido así:

  Comentario = --Not(eol)*eol

* Un literal decimal puede ser definido así:

  Literal = $D^+.D^+$

* Un identificador, que inicia con una letra, seguida por letras, dígitos o el símbolo de subrayado y que termina con una letra o con un dígito solamente y no contiene sucesivos subrayados, puede ser definido así:
  Identificador = L(L|D)*(_(L|D)+)*

* Un ejemplo mas complicado es un comentario delimitado por ## que permite un # dentro del comentario
  Comentario2 = ##((#|λ)Not(#))*##

Definiciones regulares

Si Σ es un alfabeto de símbolos básicos, entonces una definición regular es una secuencia de definiciones de la forma

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
……....
…….
$d_n \rightarrow r_n$

donde cada d_i es un nombre distinto, y cada r_i es una expresión regular sobre los símbolos de $\Sigma \cup \{d_1, d_2, \ldots, d_n\}$

Ejemplo:

dígito → 0|1|…|9
dígitos → **dígito dígito***
fracción operativa → . **dígitos**| **λ**
exponente operativo →(E(+|-|λ) **dígitos**) | λ

Autómatas Finitos y Scanners

Un **autómata finito** (FA) es usado para reconocer los tokens especificados por expresiones regulares.
Un autómata finito consiste de:

i) un conjunto finito de estados,
ii) un conjunto de transiciones de un estado a otro, rotulados con caracteres de V,
iii) un estado inicial, y
iv) un estado final.

Un autómata finito puede ser representado gráficamente usando un diagrama de transición.

$(a\ b\ (c)^+)^+$

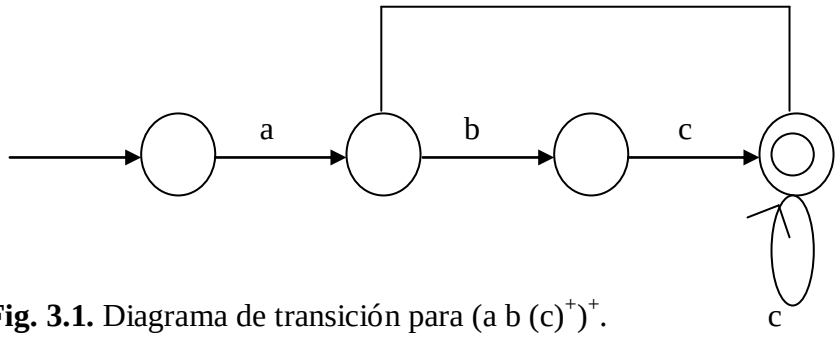
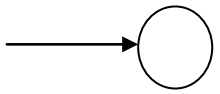


Fig. 3.1. Diagrama de transición para $(a\ b\ (c)^+)^+$.



estado de inicio

estado final

un estado

una transición

Si un autómata finito tiene una única transición para un estado y carácter dados, el autómata finito es **determinístico**. Un autómata finito determinístico es convenientemente representado en una computadora por una **tabla de transición.** Una tabla de transición T es indexada por los estados y el vocabulario de símbolos del autómata finito determinístico. Las entradas de la tabla T son estados del autómata o una bandera de error. Si estamos en el estado s y leemos el carácter c, entonces T[s] [c] será el siguiente estado a seguir, o indicar con una bandera de error que c no puede ser parte del token que se pretende leer.

Ejemplo:

La expresión regular --Not(eol)*eol que define un comentario en MICRO, puede ser traducida como

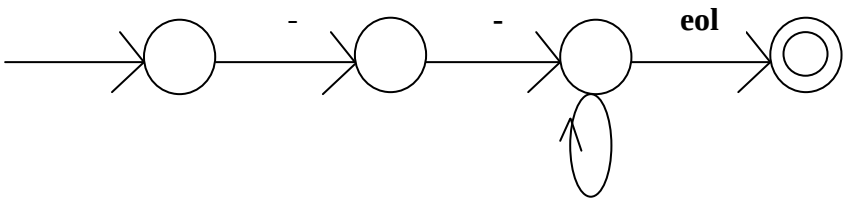


Fig. 3.2. Diagrama de transición para --Not(eol)*.

La correspondiente tabla de transición es:

Estado	Carácter		
	-	eol	Not(eol)
1	2	*	*
2	3	*	*
3	3	4	3
4	*	*	*

*: representa error

Fig. 3.3. Tabla de transición para --Not(eol)*.

Ejemplos:

1. Un literal entero en FORTRAN puede ser definido como

   RealLit = (D^+ (λ | .)) | (D^* . D^+)

   el cual corresponde al siguiente autómata finito determinístico

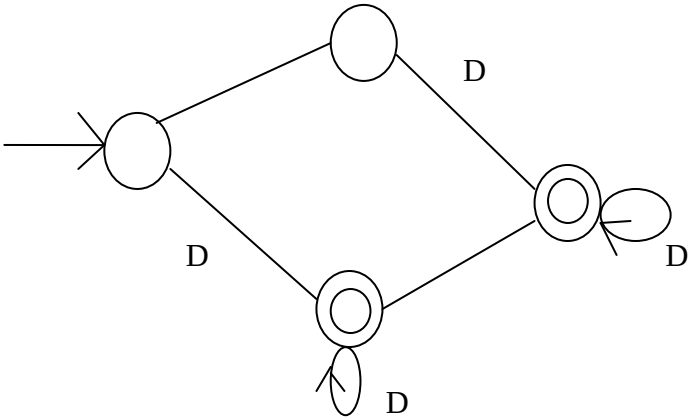


Fig. 3.4. Un autómata finito determinístico.

2. Un identificador puede definirse como

   Id = L (L | D)* (_ (L | D)$^+$)*

   el cual corresponde al siguiente autómata finito característico

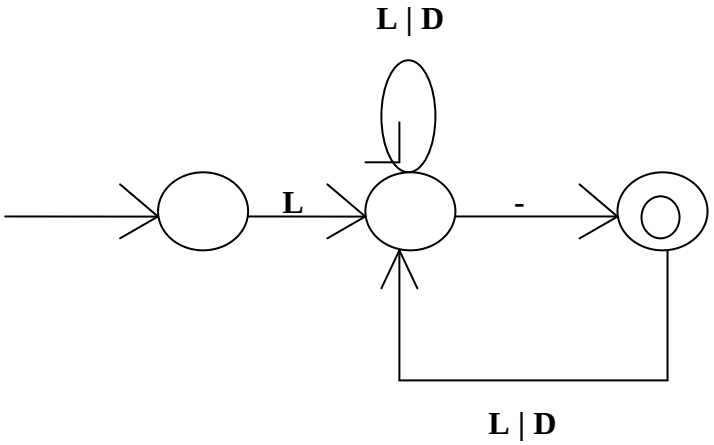
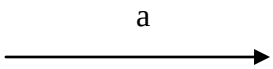


Fig. 3.5. Un autómata finito característico.

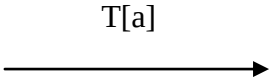
.

Es posible agregar una facilidad para producir salidas en un autómata finito; esto hace el autómata finito un traductor. A medida que los caracteres son leídos, ellos son transformados y concatenados a una cadena de salida. Para nuestra ilustración, limitamos esta transformación a guardar o borrar un carácter. Después que el token es reconocido, el input transformado puede pasarse a otra fase del compilador para futuros propósitos.

Usaremos la siguiente notación:

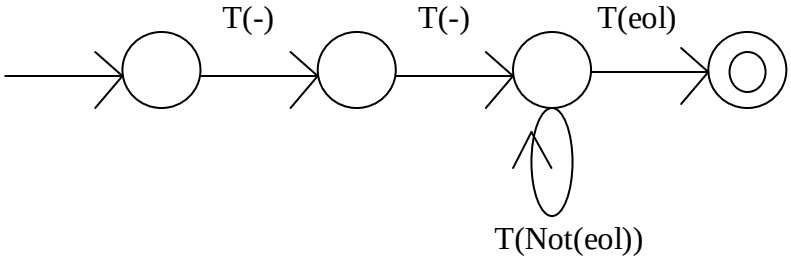


Significa guardar el carácter a en el buffer



Significa no guardar el carácter a en el buffer.

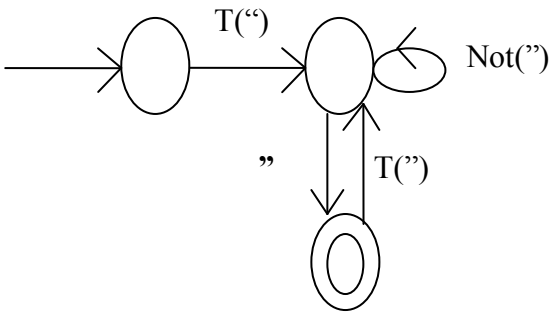
Ejemplo 3.1. Para comentarios escribiríamos:



La expresión regular para cadenas corridas es:

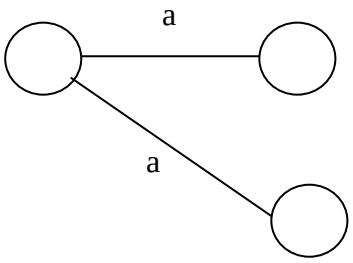
(“ (Not(“) | “”)* ”)

y su correspondiente autómata finito característico es:

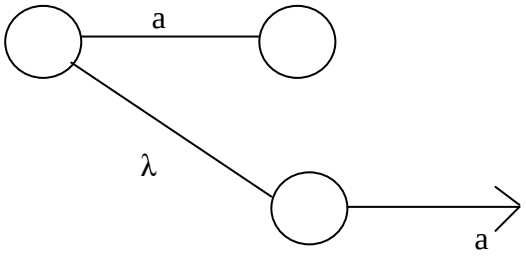


Traduciendo expresiones regulares en autómatas finitos

Expresiones regulares son equivalentes a autómatas finitos. De hecho, el trabajo principal de un generador de scanner es transformar definiciones de expresiones regulares en sus equivalentes autómatas finitos. Eso se hace primero transformando la expresión regular en autómata finito no determinístico. Después de leer un input en particular, un autómata finito no determinístico no esta obligado a realizar una única selección de cual es el próximo estado a seguir. En un autómata finito no determinístico pueden existir transiciones etiquetadas con λ.



Un autómata finito no determinístico
con dos transiciones para a



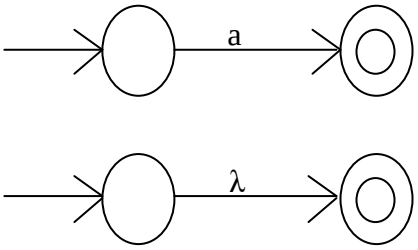
Un autómata finito no determinístico
con una transición.

El algoritmo para hacer un autómata finito determinístico de expresiones regulares se ejecuta en dos pasos:

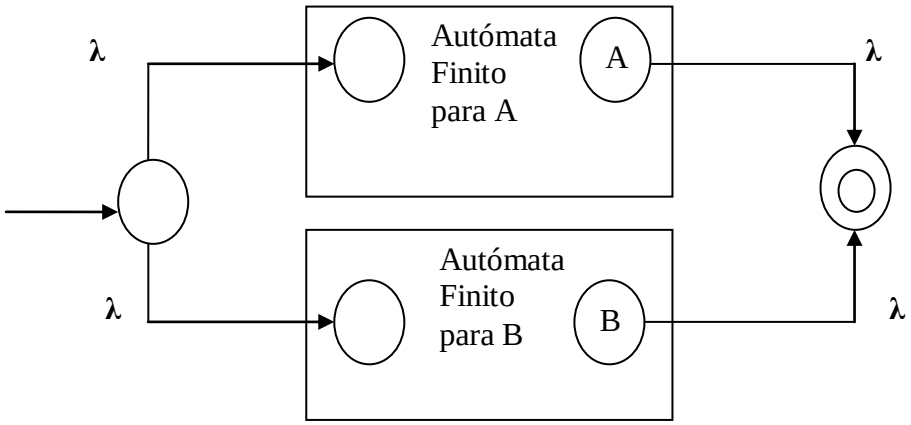
1. Transformar las expresiones regulares en un autómata finito no determinístico siguiendo las siguientes propiedades:
    * Existe un único estado final
    * El estado final no tiene sucesores
    * Cada otro estado tiene uno o dos sucesores
2. Transformar el autómata finito no determinístico en un autómata de finito determinístico.

Ejemplos:

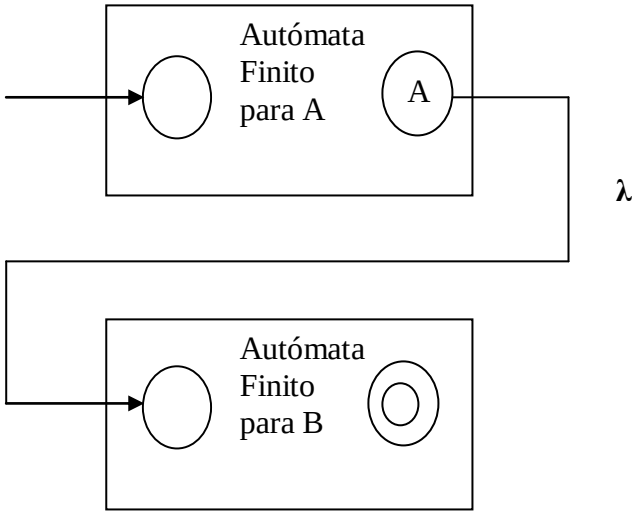
1. Autómatas para **a** y λ



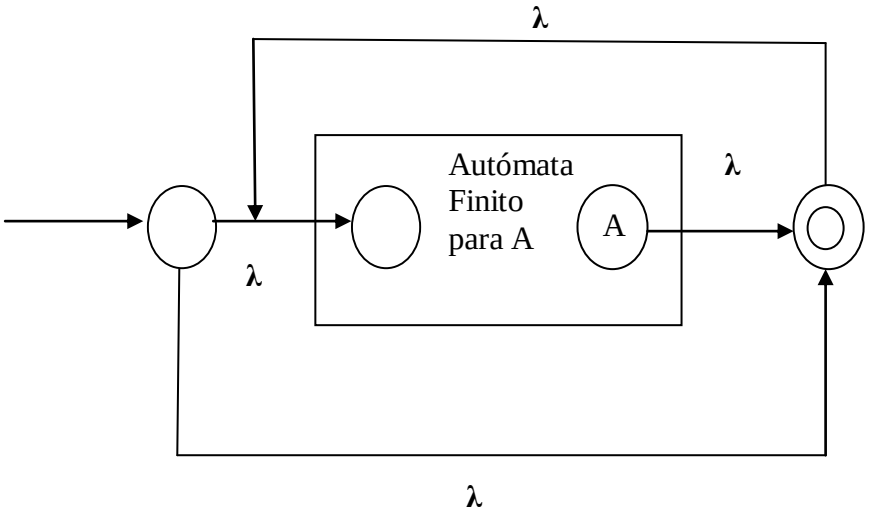
2. Un autómata finito no determinístico para A | B



3. Un autómata finito no determinístico para AB

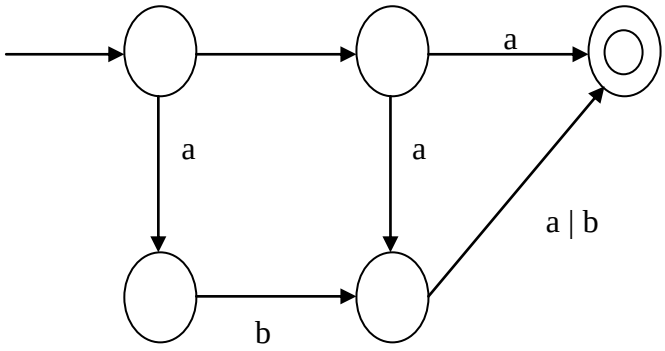


4. Un autómata finito no determinístico para A^*

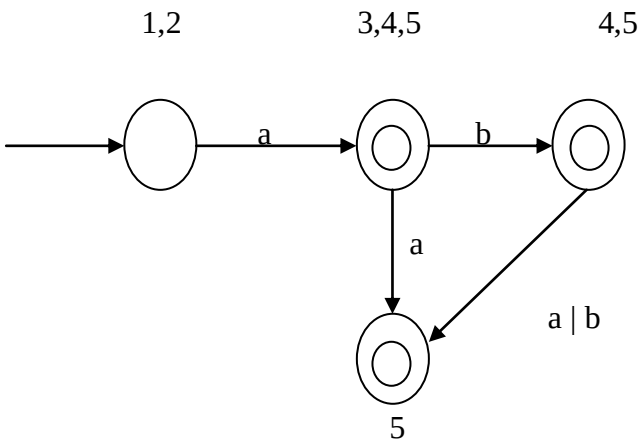


La transformación de un autómata finito no determinístico N en su equivalente autómata finito determinístico M es basada en lo que conoce como "construcción de subconjuntos". Cada estado de M corresponde a un conjunto de estados de N. La idea es que M tendrá un estado {x, y, z} después de leer una cadena de entrada sí y solo sí N puede tener cualquiera de los estados x, y o z, dependiendo de la transición que se escoja. Así M mantiene todas las posibles rutas que N puede tomar y correr ellas en paralelo.

Ejemplo 3.2. Considere el siguiente autómata finito no determinístico.

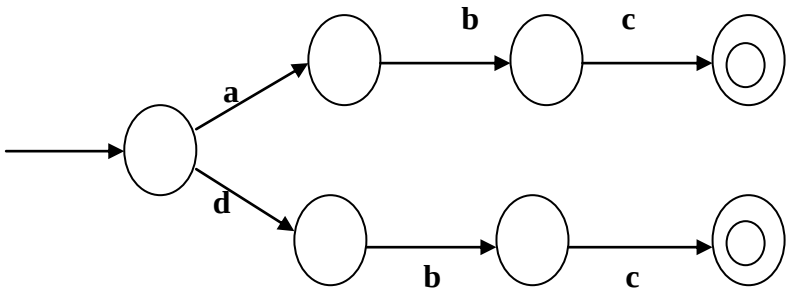


Equivalente a

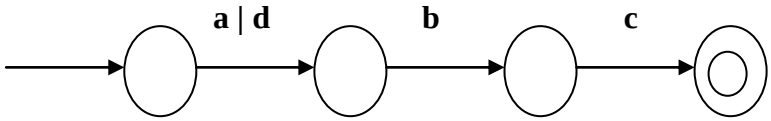


Optimización de los Autómatas Finitos

El siguiente autómata



es equivalente a:



Consideraciones prácticas

Consideraciones necesarias para construir un scanner real para lenguajes de programación.

Palabras Reservadas

- Tratar palabras reservadas como identificadores y usar una tabla separada para detectarlas.
- Organizar una tabla de excepciones como una lista de excepciones con búsqueda binaria o en una tabla hash.
- Reconocer las palabras reservadas creando distintas expresiones regulares para cada una de las palabras reservadas.

Directivos de Compilación y Listado de Líneas

- Directivos de compilación y pragmas son usados para controlar opciones de compilación (listados, optimizaciones). Si el directivo es una bandera, este puede ser extraído de un token. El comando es ejecutado y finalmente el token es borrado.

Ejemplo. Mensajes de errores y Numeración de líneas.

Cuando cada token es regresado por el scanner , su posición en la línea de salida debe ser incluida en el buffer, así, si algún error con el token es detectado, su posición debe ser marcada apuntando al token e informado el numero de la línea de programa donde se encuentra. Ejemplo:

23: if(x >< y) then y = x;

Entrada de Identificadores en la Tabla de Símbolos

En algunos scanners es común introducir un identificador a la Tabla de Símbolos inmediatamente después de su detección, si este no se encuentra de antemano en la Tabla. Independientemente si el identificador es introducido en la tabla, un puntero a la entrada de la Tabla de Símbolos debe ser devuelto por el scanner.

Información que puede almacenarse en la Tabla de Símbolos. Ejemplo, la utilización de variables:

locales y globales
en declaraciones
en el campo de un récord
como etiqueta, etc.

Finalización del scanner

El uso del token SCANEOF es de gran utilidad para verificar que el fin lógico del programa corresponde con el fin físico del mismo.

Lookahead multicaracter

Podemos generalizar los autómatas finitos para visualizar más allá del siguiente carácter del input. Este tópico es muy importante para la implementación de FORTRAN. Por ejemplo, la orden DO IO I =1, 100 es el comienzo de un ciclo, con índice I que varia de 1 a 100. La orden DO IO I=1,100 es una asignación a la variable DO IOI (los espacios en blancos no son significativos en FORTRAN).

Lookahead multicaracter es una consideración necesaria scaneando programas inválidos. Por ejemplo, 12.3e+q es un token invalido. En este caso el scanner debe regresar atrás y producir cuatro tokens. Ya que esta secuencia de tokens (12.3, 3,+, q) es valida, el parser detectara un error de sintaxis cuando este procese la secuencia. No es relevante si se escoge tratarlo como error léxico o error de sintaxis, pero en alguna fase el compilador debe detectar el error.

Recuperación de errores léxicos

Ocasionalmente el scanner detectara errores léxicos. No es razonable detener la compilación porque un error menor fue detectado, por lo tanto es necesario intentar alguna manera de recuperarse después del error. Dos maneras surgen:
*Borrar los caracteres leídos hasta ahora y recomenzar el scaneo con el siguiente carácter sin leer.
*Borrar el primer carácter leído por el scanner y reiniciar el scaneo con el carácter siguiente.

Ambas maneras son razonables. La primera es muy fácil de implementar. Es sólo resetear el scanner y empezar a escanear de nuevo. La segunda es un poco más difícil pero más segura. Esta puede ser implementada usando el mecanismo de buffering (almacenamiento).

Flex

Flex es una herramienta para generar escáners: programas que reconocen patrones léxicos en un texto. Flex lee los ficheros de entrada dados, o la entrada estándar si no se le ha indicado ningún nombre de fichero, con la descripción de un escáner a generar. La descripción se encuentra en forma de parejas de expresiones regulares y código C, llamadas *reglas.* Flex genera como salida un fichero fuente en C, **lex.yy.c,** que define una rutina **yylex().** Este fichero se compila y se enlaza con la librería **−lfl** 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.

Algunos ejemplos simples

En primer lugar veremos algunos ejemplos simples para una toma de contacto con el uso de flex. La siguiente entrada de flex especifica un escáner que siempre que encuentre la cadena "username" la reemplazará por el nombre de entrada al sistema del usuario:

```
%%
username printf( "%s", getlogin() );
```

Por defecto, cualquier texto que no reconozca el analizador léxico de flex se copia a la salida, así que el efecto neto de este escáner es copiar su fichero de entrada a la salida con cada aparición de "username" expandida. En esta entrada, hay solamente una regla. "username" es el *patrón* y el "printf" es la *acción*. El "%%" marca el comienzo de las reglas.

Aquí hay otro ejemplo simple:

```
int num_lineas = 0, num_caracteres = 0;

%%
\n ++num_lineas; ++num_caracteres;
. ++num_caracteres;
%%
 main()
{
    yylex();

    printf( "# de líneas = %d, # de caracteres. = %d\n",
    num_lineas, num_caracteres );
 }
```

Este analizador cuenta el número de caracteres y el número de líneas en su entrada (no produce otra salida que el informe final de la cuenta). La primera línea declara dos variables globales, "num_lineas" y "num_caracteres", que son visibles al mismo tiempo dentro de **yylex()** y en la rutina **main()** declarada después del segundo "%%". Hay dos reglas, una que empareja una línea nueva ("\n") e incrementa la cuenta de líneas y la cuenta de caracteres, y la que empareja cualquier caracter que no sea una línea nueva (indicado por la expresión regular ".").

Tema 4. GRAMÁTICA LIBRE DE CONTEXTO Y PARSER

Objetivos

- Define conceptos y notación de la Gramática Libre de Contexto.
- Clasifica los errores comunes de la Gramática Libre de Contexto.
- Identifica la Forma Bachus Naur Extendida.
- Aplica gramática para reconocer parser hacia arriba, abajo.
- Derivación por la derecha y por la izquierda.
- Define conceptos básicos sobre gramática y derivaciones.
- Define conjunto primero y siguiente de la gramática.

Gramáticas Libre de Contextos: Conceptos y notación.

Una CFG está definida por los cuatro siguientes componentes:

1) Un vocabulario terminal finito Vt; esto es el conjunto de tokens producido por el scanner.
2) Un conjunto de diferencia finito, símbolos intermedios, llamado el vocabulario no terminal Vn.
3) Un símbolo de inicio S ϵ Vn que inicia todas las derivaciones.
4) Un conjunto finito de reproducciones P (algunas veces llamado reglas de reescritura) de la forma
   $A \rightarrow X_1 \ldots X_m$, donde $A \in Vn$, $X_i \in Vn \cup Vt$, $1 \leq i \leq m$, $m \geq 0$.

Note que $A \rightarrow \lambda$ es una producción válida.

Abreviadamente: CFG=(Vt,Vn,S,P)

El vocabulario V de una CFG es un conjunto $Vt \cup Vn$.

El conjunto de cadenas derivables de S define el lenguaje libre de contexto de la gramática G, denotado por L(G).

Notaciones:

a,b,c,…	denotan símbolos en Vt.
A,B,C,…	denotan símbolos en Vn.
U,V,W,…	denotan símbolos en V.
α,β,γ,…	denotan símbolos en V^*.
u,v,w,…	denotan símbolos en Vt^*.

$A \rightarrow \alpha | \beta | \ldots\ldots | \xi |$ es la derivación de

$A \rightarrow \alpha$

$A \rightarrow \beta$

…….

$A \rightarrow \xi$

Si $A \rightarrow \gamma$ es una producción, entonces $\alpha A \beta \Rightarrow \alpha\gamma\beta$, donde $\Rightarrow$ denota una derivación de un paso.

Extenderemos $\Rightarrow$ a $\Rightarrow^+$, derivando en uno o más pasos, y $\Rightarrow^*$, derivando en cero o más pasos.

Si $S \Rightarrow^* \beta$, entonces β es llamado a ser una forma sentencial de la CFG. SF(G) es el conjunto de formas sentenciales de G.

Similarmente, $L(G) = \{n \in Vt^* \mid S \in \Rightarrow^+ X\}$

Note que $L(G) = SF(G) \cap Vt^*$

Esto significa que el lenguaje de G es simplemente aquellas formas senténciales de G que son cadenas terminales.

Cuando se deriva una secuencia de tokens, si más de una no terminal está presente, tenemos una opción de cual es la siguiente a expandir. Para caracterizar una secuencia de derivación, necesitamos pues especificar, en cada paso, cual no terminal está siendo expandido y que regla de producción es aplicada. Adoptaremos una convención acerca que el no terminal debe ser expandido en cada caso.

Una convención obvia es escoger el no terminal a la izquierda del lado derecho de la producción en cada paso. A esta convención se le llama derivación por la izquierda y se denota usando $\Rightarrow_{lm}$, $\Rightarrow^+_{lm}$ y $\Rightarrow^*_{lm}$.

Una forma sentencial producida por una derivación por la izquierda, es llamada forma sentencial izquierda. La secuencia de producción descubiertas por una gran mayoría de Parsers (parsers de arriba hacia abajo) son derivaciones por la izquierda, de aquí, los parsers son llamados **parsers por la izquierda.**

Ejemplo 4.1. Considere la siguiente gramática G0 que genera simple expresiones con variables y funciones.

1. E → Prefix (E)
2. E → V Tail
3. Prefix → F
4. Prefix → λ
5. Tail → + E
6. Tail → λ

La derivación por la izquierda para F(V+V) será:

E $\Rightarrow_{lm}$ Prefix (E)	por(1)
$\Rightarrow_{lm}$ F(E)	por(3)
$\Rightarrow_{lm}$ F(V Tail)	por(2)
$\Rightarrow_{lm}$ F(V + E)	por(5)
$\Rightarrow_{lm}$ F(V + V Tail)	por(2)
$\Rightarrow_{lm}$ F(V +V)	por(6)

Una alternativa a la derivación por la izquierda es la **derivación por la derecha**, también llamada **canónica**, en la cual el no terminal de la derecha es siempre expandido. Esta técnica corresponde a otra clase de parsers, los **parsers de abajo hacia arriba.**

Notaciones: $\Rightarrow_{rm}$, $\Rightarrow^{+}_{rm}$ y $\Rightarrow^{*}_{rm.}$

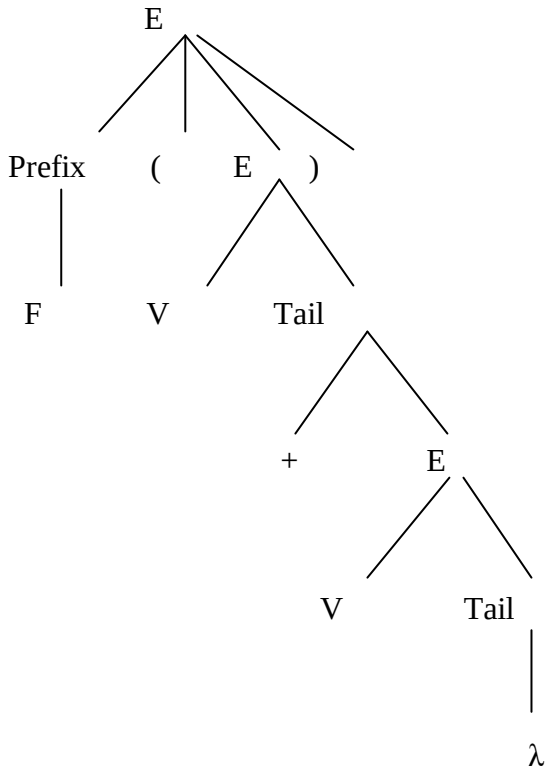
Una forma sentencial por la derecha es una forma sentencial producida vía derivación por la derecha.

Ejemplo: una derivación por la derecha de F(V+V) será:

E $\Rightarrow_{rm}$ Prefix (E)
$\Rightarrow_{rm}$ Prefix(V Tail)
$\Rightarrow_{rm}$ Prefix(V + E)
$\Rightarrow_{rm}$ Prefix(V + V Tail)
$\Rightarrow_{rm}$ Prefix(V + V)
$\Rightarrow_{rm}$ F(V + V)

Una derivación es frecuentemente representada por un **árbol de parseo**. Un árbol de parseo tiene su raíz en el símbolo de inicio **S**; sus hojas son símbolos de la gramática o λ. Los nodos interiores del árbol de parseo son no terminales.

Ejemplo: El árbol de parseo correspondiente a F(V+V) es:



Errores en CFG (Gramática Libre de Contexto).

CFGs (Gramáticas Libre de Contexto) son un mecanismo definicional. Sin embargo, ellas pueden tener errores, al igual que cualquier programa. Algunos de estos errores son fáciles de detectar, otros son más difíciles.

La notación básica de CFGs (Gramáticas Libre de Contexto.) es que iniciamos con el símbolo de comienzo y aplicamos las producciones hasta que el símbolo terminal es producido.

Considere la siguiente gramática G1:

S→A | B
A→a
B→Bb
C→c

En G1, el no terminal c no puede ser alcanzado desde S (el símbolo de inicio), y el no terminal B no deriva una cadena terminal. No terminales inalcanzables o que no derivan una cadena terminal son llamados **no usables.** No usables no terminales y producciones que las contienen pueden ser removidas de las gramáticas sin cambiar el lenguaje definido por la gramática. Una gramática que contiene no terminales no usables se llama **no reducida.** Así G1 es una gramática no reducida. Después de remover B y c, se obtiene una gramática equivalente G2, la cual es reducida:

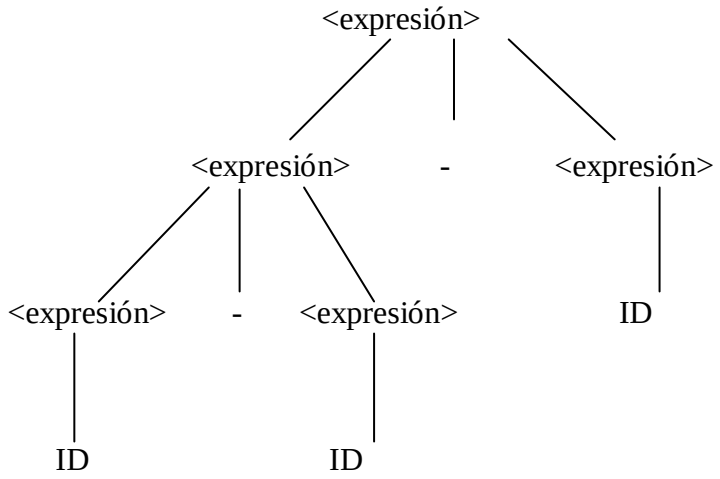
S→A
A→a

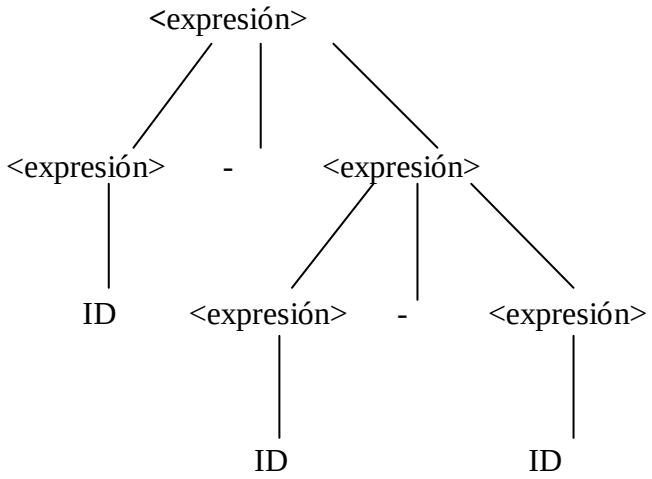
Un error más en la gramática es que algunas veces una gramática permite a un programa tener 2 o más diferentes árboles de derivación o parseo (y así una estructura no única).

Ejemplo: Considere la gramática siguiente:

<expresión> → <expresión> - <expresión>
<expresión> →ID

Esta gramática permite dos diferentes árboles de parseo para ID – ID – ID.





Gramáticas que permiten diferentes árboles de parseo para la misma cadena terminal son llamadas **ambiguas.**

Nosotros nos restringimos a usar gramática sin ambigüedades para garantizar estructuras únicas.

Algoritmos para Análisis de Gramáticas

Transformando Gramáticas en la Forma Bachus Naur Extendida.

A como se ha visto gramáticas en la forma BNE son muy útiles representando muchos lenguajes de programación porque esta forma permite definir opciones usando [,] y permite construcción opcional de listas que son definidas usando {}. Analizando gramáticas y construyendo parsers es de mucha ayuda asumir que la gramática esta en forma estándar. A continuación describimos un algoritmo para transformar gramáticas BNFE en forma estándard.

Para cada producción con opcionalidad $P = A \rightarrow \alpha [X_1 \ldots X_n] \beta$

1. –Crear un nuevo no terminal N.
2. – Reemplazar la producción P por $P' = A \rightarrow \alpha N \beta$
3. – Agregar las producciones $N \rightarrow X_1 \ldots\ldots X_n$ y $N \rightarrow \lambda$

Para cada producción con la clausura de Kleene $Q = \beta \rightarrow \gamma \{Y_1 \ldots Y_m\} \gamma$

1. Crear un nuevo no terminal M
2. Reemplazar la producción Q por $Q' = \beta \rightarrow \gamma M \gamma$
3. Agregar las producciones $M \rightarrow Y_1 \ldots Y_m M$ y $M \rightarrow \lambda$

Frecuentemente es necesario analizar las propiedades de una gramática para determinar si una gramática es parseable y, si lo es, construir las tablas que puedan ser usadas para conducir un algoritmo de parseo.

Una gramática G, es representada por un record (en C, struct). Este tiene campos llamados terminales, nonterminales, productions, start symbol.

El campo productions es un arreglo de struct, cada una de las cuales tiene campos lhs, rhs y rhs_length.

El termino vocabulary es una combinación de terminals y nonterminals.

Asi tenemos las siguientes declaraciones en C:

```
typedef int symbol;
/*símbolos en la gramática
*num_terminals, num_nonterminals y num_productions son
*determinadas por la gramática.
*max_rhs_length debe ser suficientemente grande
*/

#define vocabulary(num_terminals + num_nonterminals)

typedef struct gram{
        symbol terminals[num_terminals];
        symbol nonterminal[num_nonterminals];
        int num_productions;
struct prod{
        symbol lhs;
        int rhs_lenght;
        symbol rhs[max_rhs_length];
}productions[num_productions];
        symbol vovabulary[VOCABULARY];
}grammar;

typedef struct prod production;
typedef symbol terminal;
typedef symbol nonterminal;
```

Una de las derivaciones más comunes en una gramática es determinar cuales no terminales pueden derivar λ

Esta información es importante puesto que los no terminales que pueden derivar λ pueden desaparecer durante el parseo y por tanto pueden ser cuidadosamente manejados.

Determinar si un no terminal puede derivar λ no es enteramente trivial porque la derivación puede tomar más de un paso.

Ejemplo:

A--> BCD --> BC --> B --> λ

Para hacer esta computación, emplearemos un algoritmo interativo para marcar los no terminales que pueden derivar λ.

```
typedf short boolean;
typedf boolean marked_vocabulary[VOCABULARY];
marked_vocabulary mark_lambda(const grammar g)
{
  static marked_vocabulary derives_lambda;
  boolean rhs_derives_lambda;    /*does the rhs derive λ?*/
  symbol v;                      /*a word in the vocabulary*/
  production p;                  /*a production on the grammar*/
  int i,j;                    /*loop variables*/

   for(v=1;v<=VOCABULARY;V++)
      derives_lambda[v]=FALSE;
       /*initially, nothing is marked*/

   for(i=1;i<=g.num_productions;i++){
       p=g.productions[i];
       /*derives lambda directly*/
        if(p.rhs_length==0)
         derives_lambda[p.lhs]=TRUE;
}

    for(i=1;i<=g.num_productions;i++){
      p=g.productions[i];
     /* I want to check the productions that do not derive lambda*/
      if(!derives_lambda[p.lhs]){
        /*does each part of RHS derive λ?*/
        rhs_derives_lambda=derives_lambda[p.rhs[1]];
        for(j=2;j<=p.rhs_length;j++)
        rhs_derives_lambda = rhs_derives_lambda[p.rhs[j]];
                             &&derives_lambda[p.rhs[j]];
        if(rhs_derives_lambda)
          derives_lambda[p.lhs]=TRUE;
}
}
return derives_lambda;
}
```

Cuando construimos un parser, frecuentemente analizamos la gramática para computar el conjunto Siguiente(A), donde A es cualquier no terminal.

Informalmente Siguiente (A) es el conjunto de terminales que pueden seguir A en alguna forma sentencial. Si A aparece en la forma sentencial como el símbolo mas a la derecha, λ es incluido en Siguiente(A), significando que A puede no tener un símbolo siguiéndolo. Mas precisamente para todo A pertenece Vn.

Siguiente(A)={a ∈ Vt | S $\rightarrow^{+}$ ...Aa ...} ∪ { λ si S$\rightarrow^{+}$∞ A}

Siguiente(A) provee el lookahead que puede señalar el reconocimiento de una producción con A como su lado izquierdo.

Otro conjunto comúnmente usado construyendo parser es, Primero(alpha) que es el conjunto de todos los símbolos terminales que pueden iniciar una forma sentencial derivable de alpha. También se incluye λ si α$\rightarrow^{*}$ λ.

Formalmente:

Primero(α) = {a ∈ Vt | ∞ →a β} ∪ { λ si ∞ →λ }

Si ∞ es el lado derecho de una producción, entonces Primero(∞) contiene los símbolos terminales que inician cadenas derivables de ∞.

Representaremos el conjunto Primero de una gramática por el arreglo first_set [x], donde x es cualquier símbolo no terminal.

Similarmente nuestra representación de Siguiente será un arreglo follow_set [A], donde A es un símbolo no terminal.

Para una cadena arbitraria ∞ (mezcla la terminales y no terminales) no podemos tener los conjuntos Primero y Siguiente precomputarizados, así que usaremos un algoritmo compute_first(∞) que devuelve el conjunto de terminales definidas por Primero(∞).

```
typedef set_of_terminal_or_lambda termset;
termset follow_set[NUM_NONTERMINAL];
termset first_set[SYMBOL];
marked_vocabulary derives_lambda=mark_lambda(g);
/*mark_lambda(g) as define above */
```

```
termset compute_first(string_of_symbols alpha)
{
 int i,k;
 termset result;

 k=length(alpha);
 if(k==0)
       result= SET_OF(λ);
 else {
      result= first_set[alpha[0]];
      for(i=1;i<k&&-- pertenece first_set[alpha[i-1]]);i++)
           result= result U SET_OF(λ);
   }
    return result;
}
```

El algoritmo fill_first set() definido a continuación inicializa los conjuntos Primero. El algoritmo opera iterativamente, primero considerando producciones sencillas, y luego considerando cadenas de producciones.

```
extern grammar g;

void fill_first_set(void)
 {
   nonterminal A;
   terminal a;
   production p;
   boolean changes;
   int i,j;

   for(i=0;i<NUM_NONTERMINAL; i++) {
        A=g.nonterminals[i];
        if(derives_lambda[A])
             first_set[A]=SET_OF(λ);
          else
             first_set[A]= Ø;
}

        for(i=0;i<NUM_TERMINAL;i++){
        a=g.terminals[i];
        first_set[a]=SET_OF(a);
        for(j=0;j<NUM_TERMINAL;j++) {
               A=g.nonterminals[j];
               if(there exists a production A→a β)
                      first_set[A]=first_set[A] ∪ SET_OF(a);
         }
      }
      do  {
         changes=FALSE;
         for(i=0;i<g.num_productions;i++){
            p=g.productions[i];
            first set[p.lhs]=first_set[p.lhs] ∪ compute_first(p.rhs);
            if(first_set changed)
                   changes=TRUE;
         }

      }while (changes);
}
```

Ejemplo 4.2. Considere la siguiente gramática G3:

E→Prefix(E)
E→V Tail
Prefix→F
Prefix→ λ
Tail→+E
Tail→ λ

La ejecución de fill_first_set() produce:

Paso	E	Prefix	Tail	(	)	V	F	+
(1)	Vacio	{--}	{--}					
(2)	{V}	{F,--}	{+,--}	{(}	{)}	{V}	(F)	{+}
(3)	{V,F(}	{F,--}	(+,--)	{(}	{)}	{V}	(F)	{+}

fill_follow_set es definido a continuación:

```
void fill_follow_set(void)
{
 nonterminal A,B;
 int i;
 boolean changes;

 for(i=0;i<NUM_NONTERMINAL;i++){
  A=g.nontermaninals[i];
  follow_set[A]=Ø;
  }
  follow_set[g.start_symbol]=SET_OF(λ)
   do  {
       changes=FALSE;
       for(each production A-->alphaBeta){
            /*
            *I.e for each production and each ocurrence
            *of a nonterminal in its right-hand side.
            */
             follow_set[B]=follow_set[B]U
                 (compute_first(beta)- SET_OF(λ));
             if(λ pertenece compute_first(beta))
                  follow_set[B]=follow_set[B]U follow_set[A];
             if(follow_set[B]changed)
                   changes=TRUE;
         }
       }while(changes);
 }
```

El algoritmo localiza las ocurrencias de no terminales en una producción y luego computa los valores de Primero para la producción sufijo siguiendo el no terminal.
Ejemplo de la ejecución de fill_follow_set:

Paso			
	E	Prefix	Tail
(1)Inicialización	{λ}	Ø	Ø
(2)Prefix en Prod 1	{ λ}	{ (}	Ø
(3)E en prod 1	{ λ,)}	{ (}	Ø
(4)Tail en prod 2	{ λ,)}	{ (}	{ λ,)}

Ejemplo 4.3. Considere la siguiente gramática G4:

S→aSe
S→B
B→bBe
B→C
C→cCe
C→d

Ejecución de fill_first_set:

Paso								
	S	B	C	a	B	c	d	e
(1) Primer loop	Ø	Ø	Ø					
(2) Segundo loop	{a}	{b}	{c,d}	{a}	{b}	{c}	{d}	{e}
(3) Tercer loop, prod 2	{a,b}	{b}	{c,d}	{a}	{b}	{c}	{d}	{e}
(4) 4to loop, prod 4	{a,b}	{b,c,d}	{c,d}	{a}	{b}	{c}	{d}	{e}
(5) 5to loop, prod 2	{a,b,c,d}	{b,c,d}	{c,d}	{a}	{b}	{c}	{d}	{e}

Ejecución de fill_follow_set:

Paso			
	S	B	C
(1) Inicialización	{λ}	Ø	Ø
(2) Proceso S, prod 1	{e,λ}	Ø	Ø
(3) Proceso B, prod 2	{e,λ}	{e,λ}	Ø
(4) Proceso B, prod 3	no changes		
(5) Proceso C, prod 4	{e,λ}	{e,λ}	{e,λ}
(6) Proceso C, prod 5	no changes		

Primero y Siguiente pueden ser generalizados para incluir cadenas de lobgitud K en vez de longitud 1. Primero $_k(\infty)$ es el conjunto de k_simbolos terminales que pueden seguir a A en alguna forma sentencial.

Parsers y Reconocedores

Asumamos que de alguna manera conocemos que una gramática es no ambigua. Dada una cadena de entrada como una secuencia de tokens, podríamos preguntarnos: ¿Es ésta entrada sintácticamente válida? (Esto es: ¿puede ésta ser generada por la gramática)? Un algoritmo que hace este test de valor booleano es llamado un **reconocedor.**

Nosotros podemos también requerir más del algoritmo y preguntar: ¿Es la entrada válida, y si lo es, cual es su estructura (árbol de parseo)? Un algoritmo que responde estas preguntas más generales es llamado **parser.** Ya que planeamos usar las estructuras de un lenguaje para construir un compilador, estaremos especialmente interesados en parsers.

Existen dos técnicas de parseo. La primera, que incluye la técnica recursiva descendente, es llamada de **arriba hacia abajo** (Top-Down). Un parser es considerado de arriba hacia abajo si este "descubre" el árbol de parseo correspondiente a la secuencia de tokens iniciando en la parte de arriba del árbol (el símbolo de inicio) y después expandiendo este (vía predicciones) en una manera de profundidad primero. Un parser de arriba hacia abajo corresponde a atravesar el árbol de parseo en preorden.

Una gran variedad de técnicas de parseo toman un estudio diferente. Estos pertenecen a la clase de parsers de **abajo hacia arriba.** A como su nombre lo sugiere, un parser de abajo hacia arriba (Botton_Up) "descubre" la estructura del árbol de parseo comenzando desde abajo (las hojas del árbol, las cuales son símbolos terminales) y determinando las producciones usadas para generar las hojas. Después las producciones usadas para generar los padres inmediatos de las hojas son descubiertas. El parser continúa hasta que alcanza la producción usada para expandir el símbolo de inicio. A este punto el árbol de parseo ha sido completamente determinado.

Árbol de parseo

Un parser de abajo hacia arriba corresponde a atravesar el árbol de parser en postorden.

Ejemplo 4.4. Considere la siguiente gramática, la cual genera el esqueleto de la estructura de bloque de un lenguaje de programación.

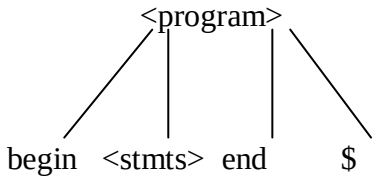
<program> →begin <stmts> end $
<stmts> →<stmt>;<stmts>
<stmts> →λ
<stmt> →Simple stmt
<stmt> → beging <stmts> end

Input: begin Simple stmt; Simple stmt; end $

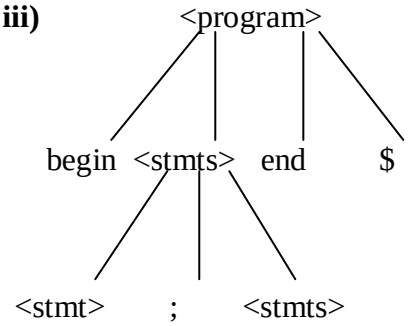
A. Parseo de arriba hacia abajo

i) <program>

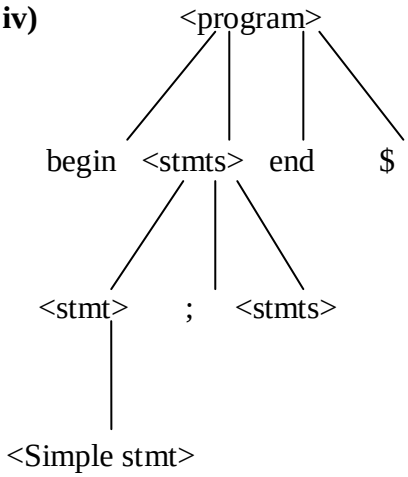
ii)



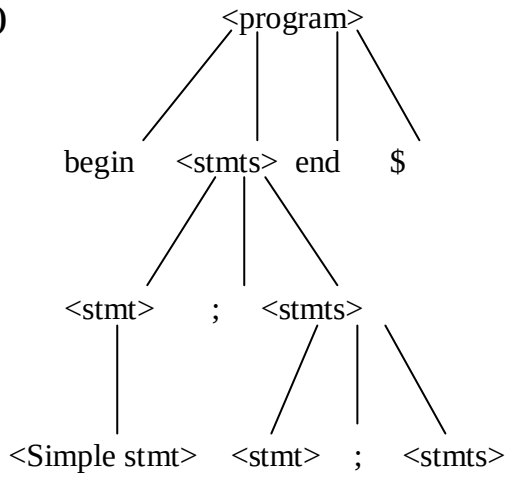
iii)



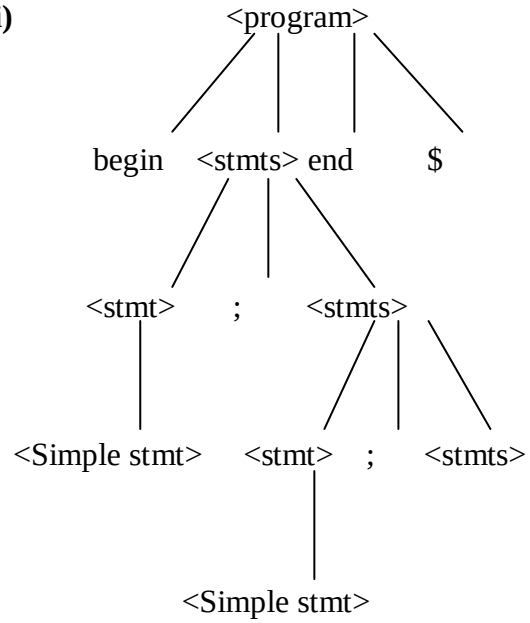
iv)

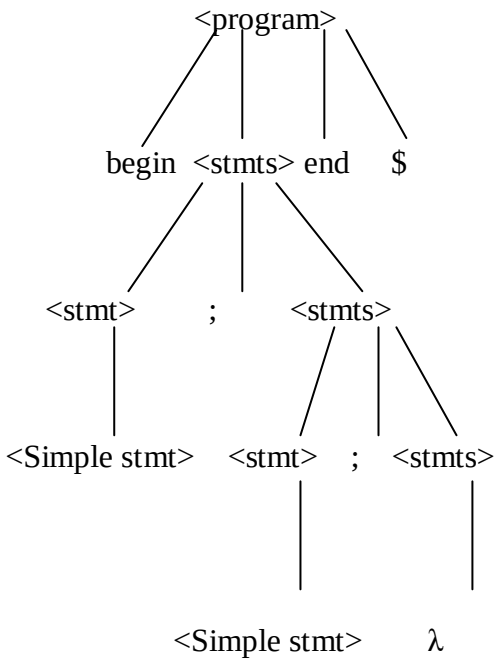

<program>
begin
<stmts>
end
$
<stmt>
;
<stmts>
<Simple stmt>

v)


<program>
begin
<stmts>
end
$
<stmt>
;
<stmts>
<Simple stmt>
<stmt>
;
<stmts>

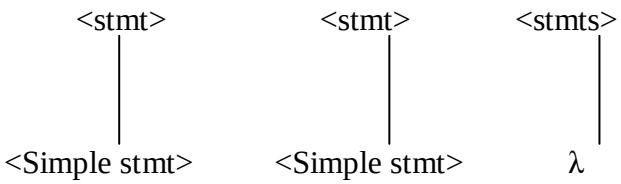
vi)


<program>
begin
<stmts>
end
$
<stmt>
;
<stmts>
<Simple stmt>
<stmt>
;
<stmts>
<Simple stmt>

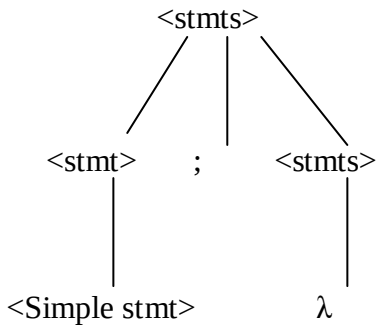

<program>
begin
<stmts>
end
$
<stmt>
;
<stmts>
<Simple stmt>
<stmt>
;
<stmts>
<Simple stmt>
λ

B. Parseo de abajo hacia arriba

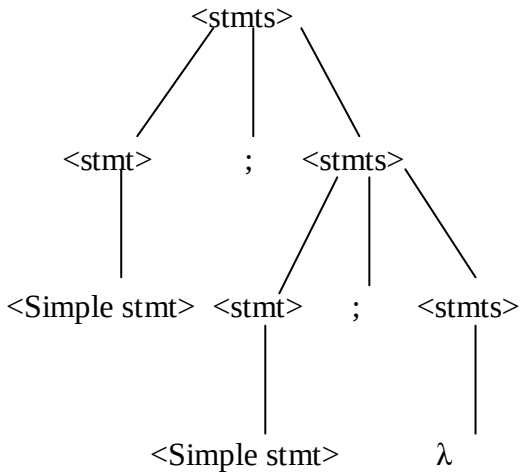
i)



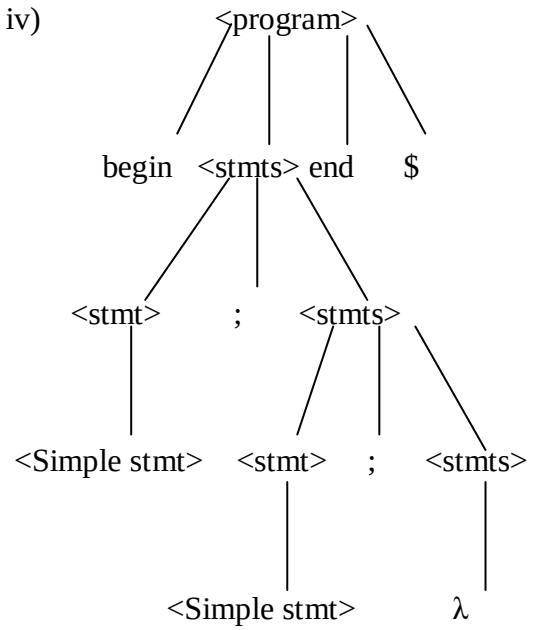
ii)



iii)



iv)



Las producciones predecidas por un parser de arriba hacia abajo representan una derivación por la izquierda. La secuencia de producciones reconocidas por un parser de abajo hacia arriba determina un parseo por la derecha.

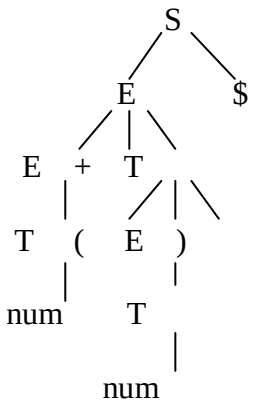
Ejemplo 4.5.

Gramática G5:

1. S→E $
2. E→E + T
3. E→T
4. T→(E)
5. T→num

Input: num + (num) $

A. **Árbol de parseo**



B. **Derivación por la derecha**

S→ E $ → E + T $ → E + (E) $
→ E + (T) $ →E + (num) $
→ T + (num) $ → num + (num) $

C. **Derivación por la izquierda**

S → E $ → E + T $ → T + T $
→ num + T $ → num + (E) $
→ num + (T) $ → num + (num) $

D. **Parseo de abajo hacia arriba (LR Parser o Shift Reduce Parser)**

PARSER STACK	OPERACIÓN	INPUT
Empty	shift	num + (num) $
num	reduce 5	+ (num) $
T	reduce 3	+ (num) $
E	shift	+ (num) $
E +	2 shifts	(num) $
E + (num	reduce 5	) $
E + (T	reduce 3	) $
E (E	shift	) $
E + (E)	reduce 4	$
E+T	reduce 2	$
E	shift	$
E $	reduce 1	Vacio
	=Aceptar !	

Primero y Siguiente

Se facilita la construcción de un analizador sintáctico predictivo con dos funciones asociadas a una gramática G. Estas funciones, PRIMERO y SIGUIENTE, permiten rellenar, siempre que sea posible, las entradas de una tabla de análisis sintáctico predictivo para G. También se pueden utilizar los conjuntos de componentes léxicos de sincronización durante la recuperación de errores en modo de pánico.

Algoritmo 4.1. Construcción de una tabla de análisis sintáctico predictivo.

Entreda. Una gramática *G*.

Salida. La tabla de análisis sintáctico *M*.

Método.

1. Para cada producción $A \rightarrow \alpha$ de la gramática, dense los pasos 2 y 3.
2. Para cada terminal *a* de PRIMERO(α), añadese $A \rightarrow \alpha$ *M*[*A*,*a*].
3. Si ϵ está en PRIMERO(α), añadese $A \rightarrow \alpha$ a *M*[*A*,*b*] para cada terminal b de SIGUIENTE (A). Si ϵ esta en PRIMERO(α) Y $ esta en SIGUIENTE(*A*), añadese $A \rightarrow \alpha$ a *M*[*A*,$].
4. Hágase que cada entrada no definida de *M* sea **error**.

Gramáticas LL(1)

Se puede aplicar el algoritmo 4.1 a cualquier gramática *G* para producir una tabla de análisis sintáctico *M*. Sin embargo, para algunas gramáticas, *M* puede tener algunas entradas con definiciones múltiples. Por ejemplo, si *G* es recursiva por la izquierda o ambigua, entonces *M* tendrá al menos una entrada con definición múltiple.

Ejemplo 4.6. Considérese la gramática G6:

$P \rightarrow iEtPP' | a$
$P' \rightarrow eP | \epsilon$
$E \rightarrow b$

En la figura 4.1 se muestra la tabla de análisis sintáctico para esta gramática

NO TERMINAL	SIMBOLO DE ENTRADA					
	A	B	E	I	T	$
P	P→a					
P'			P'→ϵ P'→eP			P'→ϵ
E		E→b				

Fig. 4.1. Tabla de análisis sintáctico M para la gramática G6.

Una gramática cuya tabla de análisis sintáctico no tiene entradas con definiciones múltiples se define como *LL*(1). La primera "L" de LL(1) representa (por *left*, en ingles, *izquierda*) el examen de la entrada de izquierda a derecha, la segunda "L" representa una derivación por la izquierda, y el "1" es por utilizar un símbolo de entrada de examen por anticipado a cada paso para tomar las decisiones de la acción en el análisis sintáctico. Se puede demostrar que el algoritmo 4.1 produce para toda gramática *G* en forma LL(1) una tabla de análisis sintáctico que analiza todas y exclusivamente, las frases de *G*.

Las gramáticas LL(1) tiene varias propiedades distintivas. Ninguna gramática ambigua o recursiva por la izquierda puede ser LL(1). También se puede demostrar que una gramática *G* es LL(1) si, y solo si, cuando $A \rightarrow \alpha \mid \beta$ sean dos producciones distintas de *G* se cumplan las siguientes condiciones:

1. Para ningún terminal *a* tanto α como β derivan a la vez cadenas que comiencen con *a*.
2. A lo sumo una de α y β puede derivar la cadena vacía.
3. Si $\beta \rightarrow \epsilon$, α no deriva ninguna cadena que comience con un terminal en SIGUIENTE(*A*).

Recuperación de errores en el análisis sintáctico predictivo

Durante el análisis sintáctico predictivo se detecta un error cuando el terminal de la cima de la pila no concuerda con el siguiente símbolo de entrada o cuando el no terminal *A* esta en la cima de la pila, *a* es el siguiente símbolo de entrada, y la entrada *M*[*A*,*a*] de la tabla de análisis sintáctico esta vacía.

La recuperación en modo pánico se basa en la idea de saltarse símbolos de la entrada hasta que aparezca un componente léxico que pertenezca a un conjunto seleccionado de componentes léxicos de sincronización. Su efectividad depende de la elección del conjunto d sincronización. Los conjuntos deben elegirse de forma que el analizador sintáctico se recupere con rapidez de los errores con más probabilidades de ocurrir en la práctica. Algunas técnicas heurísticas son las siguientes:

1. Como punto de partida, se pueden colocar todos los símbolos de SIGUIENTE(*A*) dentro del conjunto de sincronización para el no terminal A. Si se saltan componentes léxicos hasta encontrar un elemento de SIGUIENTE (*A*) y se saca a *A* de la pila, es probable que el análisis sintáctico puede continuar.

2. No es suficiente usar SIGUIENTE(A) como conjunto de sincronización para A. por ejemplo, si los símbolos de punto y coma terminan las proposiciones, como en C, entonces las palabras clave que comienzan proposiciones pueden no aparecer en el conjunto SIGUIENTE del no terminal que genera las expresiones. Por tanto, un punto y coma que falte después de una asignación puede dar como resultado que se salte la palabra clave que inicia la siguiente proposición. A menudo hay una estructura jerárquica en las construcciones de un lenguaje; por ejemplo, las expresiones aparecen dentro de proposiciones, las cuales aparecen dentro de bloques, y así sucesivamente. Se pueden a; adir al conjunto de sincronización de una construcción de menor jerarquía los símbolos que inician las construcciones de mayor jerarquía. Por ejemplo, se pueden agregar palabras clave que comienzan proposiciones a los conjuntos de sincronización para los no terminales que generen expresiones.

3. Si se añaden símbolos de PRIMERO(*A*) al conjunto de sincronización para el no terminal *A*, entonces se puede continuar el análisis sintáctico según *A* si aparece en la entrada un símbolo de PRIMERO(*A*).

4. Si un no terminal puede generar la cadena vacía, se puede usar la producción que derive a **ϵ** como alternativa por omisión. Esto puede posponer alguna detección de errores pero no la omisión de un error. Este método reduce el número de no terminales que hay que considerar durante la recuperación del error.

5. Si no se puede emparejar un terminal de la cima de la pila, una idea sencilla es sacar el terminal, emitir un mensaje que indique que se inserto el terminal y continuar el análisis. En realidad, este método considera al conjunto de sincronización de un componente léxico como si estuviera compuesto por todos los otros componentes léxicos.

Análisis sintáctico ascendente

En esta sección se introduce un estilo general de análisis sintáctico ascendente, conocido como análisis sintáctico por desplazamiento y reducción.
El análisis sintáctico por desplazamiento y reducción intenta construir un árbol de análisis sintáctico para una cadena de entrada que comienza por las hojas (el fondo) y avanza hacia la raíz (la cima). Se puede considerar este proceso como de "reducir" una cadena w al símbolo inicial de la gramática. En cada paso de *reducción* se sustituye una subcadena determinada que concuerda con el lado derecho de una producción por el símbolo del lado izquierdo de dicha producción y si en cada paso se elige correctamente la subcadena, se traza una derivación por la derecha en sentido inverso.

Ejemplo 4.7. Considérese la gramática G7:

S→aABe
A→Abc | b
B→d

La frase *abbcde* se puede reducir a *S* por los siguientes pasos:
abbcde
aAbcde
aAde
aABe
S

Se examina *abbcde* buscando una subcadena que concuerde con el lado derecho de alguna producción. Las subcadenas *b* y *d* sirven. Elijase la *b* situada más a la izquierda y sustitúyase por *A*, el lado izquierdo de la producción *A→b*; así se obtiene la cadena *aAbcde*. A continuación, las subcadenas *Abc*, *b* y *d* concuerdan con el lado derecho de alguna producción. Aunque *b* es la subcadena situada más a la izquierda que concuerda con el lado derecho de una producción, se elige sustituir la subcadena *Abc* por *A*, que es lado derecho de la producción *A→ Abc*. Se obtiene ahora aAde. Sustituyendo después d por *B*, que es el lado izquierdo de la producción *B→d*, se obtiene *aABe*. Ahora se puede sustituir toda esta cadena por *S*. por tanto, mediante una secuencia de cuatro reducciones se puede reducir abbcde a *S*. De hecho, estas reducciones trazan la siguiente derivación por la derecha en orden inverso:
S→ aABe→aAde→aAbcde→abbcde

Mangos

Informalmente, un "mango" de una cadena es una subcadena que concuerda con el lado derecho de una producción y cuya reducción al no terminal del lado izquierdo de la producción representa un paso a lo largo de la inversa de una derivación por la derecha.

En muchos casos, la subcadena situada más a la izquierda β que concuerda con el lado derecho de alguna producción *A→*β no es un mango, porque una reducción por la producción *A→*β produce una cadena no reducible al símbolo no inicial. En el ejemplo anterior, si se sustituyera b por A en la segunda cadena aAbcde se obtendría la cadena aAAcde que no se puede reducir posteriormente a *S*. Por esta razón se debe dar una definición más precisa de un mango.

Formalmente, un mango de una forma de frase derecha γ es una producción *A→*β y una posición de γ donde la cadena β podría encontrarse y sustituirse por *A* para producir la forma de frase derecha previa en una derivación por la derecha de γ. Es decir, si *S→ aAw→* αβ*w*, entonces *A→*β si la posición que sigue de α es un mango de αβ*w*. La cadena *w* a la derecha del mango contiene solo símbolos terminales. Obsérvese que dice "un mango" en lugar de "el mango", porque la gramática podría ser ambigua, con más de una derivación por la derecha de αβ*w*. Si una gramática no es ambigua, entonces toda forma de frase derecha de la gramática tiene exactamente un mango.

En el ejemplo anterior, abbcde es una forma de frase derecha cuyo mango es $A \rightarrow b$ en la posición 2. Del mismo modo, *aAbcde* es una forma de frase derecha cuyo mango es $A \rightarrow Abc$ en la posición 2. Algunas veces se dice "la subcadena β es un mango de αβ*w*" si están claras la posición de β y la producción $A \rightarrow \beta$ que se tienen en mente.

En la siguiente figura se representa el mango $A \rightarrow \beta$ en el árbol de análisis sintáctico de una forma de frase derecha αβ*w*. El mango representa al subárbol completo situado más a la izquierda que consta de un nodo y todos sus hijos. En la figura de abajo, *A* es el nodo interior situado más abajo y mas a la izquierda con todos sus hijos en el árbol. Se puede considerar como "poda del mango", es decir, eliminación de los hijos de *A* del árbol de análisis sintáctico.

Ejemplo 4.8. Considérese la siguiente gramática G8:

(1) E→E + E
(2) E→E * E
(3) E→(E)
(4) E→**id**

Y la derivación por la izquierda

E→<u>E + E</u>
→E = <u>E * E</u>
→E + E * **<u>id$_3$</u>**
→ E + **<u>id$_2$</u> * id$_3$**
→**<u>id$_1$</u>** + **id$_2$** * **id$_3$**

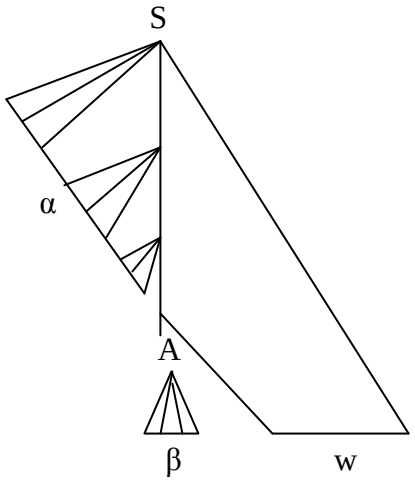


Fig. 4.2. El mango A→β en el árbol de análisis sintáctico para αβw.

Para facilitar la notación se han puesto subíndices a los símbolos **id** y se ha subrayado un mango de cada forma de frase derecha. Por ejemplo, $\mathbf{id}_1$ es un mango de la forma de frase derecha $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$, porque id es el lado derecho de la producción E→**id**, y sustituir $\mathbf{id}_1$ por *E* produce la forma de frase derecha previo $E + \mathbf{id}_2 * \mathbf{id}_3$.
Obsérvese que la cadena que aparece a la derecha de un mango contiene solo símbolos terminales.

Puesto que la gramática G8 es ambigua, hay otra derivación por la derecha de la misma cadena:

$E \rightarrow \underline{E * E}$
$\rightarrow E * \underline{\mathbf{id}_3}$
$\rightarrow \underline{E + E} * \mathbf{id}_3$
$\rightarrow E + \underline{\mathbf{id}_2} * \mathbf{id}_3$
$\rightarrow \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3$

Considérese la forma de frase derecha $E + E * \mathbf{id}_3$. En esta derivación, $E + E$ es un mango de $E + E * \mathbf{id}_3$, mientras de $\mathbf{id}_3$ por sí mismo es un mango de esta misma forma de frase derecha según la derivación anterior.

Tablas de análisis sintáctico SLR

A continuación se muestra como construir las funciones de acción e *ir_a* del análisis sintáctico SLR a partir del autómata finito determinista que reconoce prefijos viables. Este algoritmo no producirá únicamente tabla de acción definida para todas las gramáticas pero funcionara correctamente con muchas gramáticas para lenguajes de programación. Dada una gramática *G*, se aumenta *G* para producir *G'* y a partir de *G'* se construye C, la colección canoníca de conjunto de elemento para G'. se construye acción, la función de acciones del analizador sintáctico, e *ir_a*, la función de transiciones de estados, a partir de C utilizando el siguiente algoritmo. El algoritmo exige que se conozca SIGUIENTE (*A*) para cada no terminal *A* de una gramática.

Algoritmo 4.2. Construcción de una tabla de análisis sintáctico SLR.

Entrada. Una gramática aumentada G'.

Salida. Las función acción e ir_a de la tabla de análisis sintáctico SLR para *G'*.

Método.

1. Constrúyase $C = \{I_0, I_1, \ldots In\}$, la colección de conjunto de elemento LR(0) para *G'*.
2. El estado *i* se construye a partir de *Ii*. Las acciones de análisis sintáctico para el estado *i* se determinan como sigue:
   a) Si $[A \rightarrow \alpha.a\beta]$ está en *Ii* e ir_a (*Ii*,a)=I*j*, entonces asígnese "desplazar j" a acción [i,a]. Aquí, a debe ser un terminal.

b) Si [A→α.] está en *Ii*, entonces asígnese "reducir *A*→α" a acción [*i, a*] para toda a en SIGUIENTE(*A*); aquí, A puede no ser *S'*.
c) Si [*S'*→*S*.] está en *Ii*, entonces asígnese "aceptar" a acción [*i*, $].

Si las reglas anteriores generan acciones contradictorias, se dice que la gramática no es SLR(1). El algoritmo no consigue en este caso producir un analizador sintáctico.

3. Las transiciones ir_a para el estado i se construyen para todos los no terminales A utilizando la regla: si *ir_a*(*Ii*,*A*)=I*i*, entonces ir_a[*i*,*A*]=*j*.
4. Todas las entradas no definidas por las reglas 2 y 3 son consideradas "error".
5. El estado inicial del analizador es el construido a partir del conjunto de elementos que contiene [*S'*→.*S*].

La tabla de análisis sintáctico formada por las funciones de acción e *ir_a* del análisis sintáctico determinada por el algoritmo anterior se denomina *tabla SLR(1)*para G. un analizador sintáctico LR que use la tabla SLR(1) para G se denomina analizador sintáctico SLR(1) para G, y una gramática que tenga una tabla de análisis sintáctico SLR(1) se denomina SLR(1). Normalmente, se omite el "(1)" después de "SLR", ya que no se consideran los analizadores sintácticos con más de un símbolo de examen por anticipado.

Construcción de tablas de análisis sintáctico LR canónico.

A continuación se introduce la técnica más general para construir una tabla de análisis sintáctico LR a partir de una gramática. Recuérdese que en el método SLR, el estado *i* pide una reducción en A→α si el conjunto de elementos I*i* contiene el elemento [A→α] y *a* esta en SIGUIENTE(A). Sin embargo, a veces, cuando el estado *i* aparece en la cima de la pila, el prefijo viable βα de la pila es tal que βA no puede ir seguida de una forma de frase derecha. Por tanto, la reducción por A→α seria no valida con la entrada *a*.

Algoritmo 4.3. Construcción de los conjuntos de elementos LR(1).

Entrada. Una gramática aumentada *G'*.

Salida. Los conjuntos de elementos LR(1) que son el conjunto de elementos válido para uno o más prefijos viables de *G'*.

Método. Los procedimientos *cerradura* e *ir_a* y la rutina principal de *elementos* para construir los conjuntos de elementos se muestran en el siguiente fragmento de código.

Construcción de conjunto de elementos LR(1) para la gramática *G'*.

```
function cerradura (I);
begin
      repeat
            for cada elemento [A→α . Bβ,a] en I,
             cada producción B→γ en G'
```

```
            y cada terminal b en PRIMERO (βa)
            tal que [B→ . γ,b]a no esté en I do
                añadir [B → . γ,b] a I;
        until no se puedan añadir más elementos a I;
        return I
end;
function ir_a (I, X);
begin
            sea J el conjunto de elementos[A→αX.β,a] tal que
            [A→α.Xβ,a] este en I.
        return cerradura (J)
end;
procedure elementos (G');
begin
        C := {cerradura({[S'→.S,$]})};
        repeat
            for cada conjunto de elementos I en C y cada símbolo
            gramatical X tal que ir_a (I,X) no esté vacio y no esté en C do
                añadir ir_a (I,X) a C
        until no se puedan añadir más conjunto de elementos a C
end
```

Recuperación de errores en el análisis sintáctico LR

Un analizador sintáctico LR detectará un error cuando consulte la tabla de acciones de análisis sintáctico y encuentre una entrada de error. Los errores nunca se detectan consultando la tabla de transicciones ir_a. A diferencia de un analizador sintáctico por precedencia de operadores, un analizador LR anunciará un error tan pronto como no haya una continuación válida para la parte de entrada examinada hasta entonces. Un analizador sintáctico LR canoníca nunca hará ni una sola reducción antes de anunciar un error. Los analizadores SLR y LALR pueden hacer varias reducciones antes de anunciar un error, pero nunca desplazaran un símbolo de entrada erróneo a la pila.

En el análisis sintáctico LR, se puede implantar una recuperación de errores en modo de pánico como sigue. Se examina la pila hasta encontrar un estado S con un valor de ir a para un determinado no terminal A. Entonces se desechan cero o más símbolos de entrada hasta encontrar un símbolo *a* que pueda seguir legalmente a *A*. Entonces el analizador sintáctico mete en la pila el estado ir_a[*s*,*A*] y prosigue normalmente el análisis sintáctico. Puede haber más de una opción para el no terminal *A*. Generalmente, estas serian no terminales que representan las partes más importante de un programa, como una expresión, una proposición o un bloque. Por ejemplo, si *A* es el no terminal prop, *a* puede ser el símbolo de punto y como o **end**.

Este método de recuperación intenta aislar la frase que tiene el error sintáctico. El analizador determina que una cadena derivable de *A* tiene un error. Parte de esta cadena ya ha sido procesada, y el resultado de este procesamiento es una secuencia de estados en el tope de la pila. El resto de la cadena aun está en la entrada, y el analizador intenta saltar el resto de la cadena buscando un símbolo en la entrada que pueda seguir

legítimamente a *A*. Eliminando estados de la pila, saltando la entrada e introduciendo ir_a[s,*A*] en la pila, el analizador supone que ha encontrado un caso de *A* y continua el análisis normal.

La recuperación a nivel de frase se implanta examinando cada entrada de error en la tabla de análisis sintáctico LR y decidiendo, basándose en el uso del lenguaje, los errores de los programadores que mas probablemente darían lugar a dicho error. Entonces se puede construir un procedimiento de recuperación apropiado; presumiblemente, el tope de la pila o los primeros símbolos de entrada o ambos se modificarían de la forma adecuada a cada entrada de error.

Comparado con los analizadores sintácticos por procedencia de operadores, el diseño de rutinas específicas para el manejo de errores para un analizador LR es relativamente sencillo. Por ejemplo, no hay que preocuparse por las reducciones defectuosas; cualquier reducción pedida por un analizador LR es correcto. Por tanto, se puede llenar cada entrada en blanco en el campo de acción con un apuntador a una rutina de error que realizara una acción apropiada elegida por el diseñador del compilador. Las acciones pueden incluir la inserción o eliminación de símbolos de la pila, de la entrada o de ambas, o la alternación y transposición de símbolos de entrada, al igual que para el analizador sintáctico por precedencia de operadores. Lo mismo que para ese analizador, se deben elegir las opciones de modo que el analizador LR no pueda caer en un lazo infinito. A este respecto, es suficiente con una estrategia que garantice que al menos un símbolo de entrada será eliminado o quizá desplazado, o que en algún momento la pila se reducirá si se ha alcanzado el final de la entrada. se debe evitar extraer un estado de la pila que abarque un no terminal, porque esta modificación elimina de la pila que abarque un no terminal, porque esta modificación elimina de la pila una construcción que ya se había analizado con éxito.

Si α es una cadena de símbolos gramaticales, se considera PRIMERO(α) como el conjunto de terminales que inician las cadenas derivadas de α. Si α -->Є, entonces Є también está en PRIMERO (α).
Se define SIGUIENTE (A), para el no terminal A, como el conjunto de terminales a que pueden aparecer inmediatamente a la derecha de A en algunas forma de frase, es decir, el conjunto de terminales a tal que haya una derivación de la forma S→ αAaβ para algún α y β. Obsérvese que en algún momento de le derivación pudieron haber existido símbolos entre A y a, pero si así fue, derivaron a ϵ y desaparecieron. Si A puede ser el símbolo situado más a la derecha en una forma de frase, entonces $ esta en SIGUIENTE(A).

Para calcular PRIMERO(X) para todos los símbolos gramaticales X, aplíquense las reglas siguientes hasta que no se puedan añadir más terminales o Є a ningún conjunto PRIMERO.

1. Si X es terminal, entonces PRIMERO(X) es {X}.
2. Si X → ϵ es una producción, entonces añádase ϵ a PRIMERO(X).
3. Si X es no terminal y X → $Y_1Y_2\ldots Y_K$ es una producción, entonces pónganse a en PRIMERO (X) si, para alguna *i*, a esta en PRIMERO (Y_i) y ϵ esta en todos los PRIMERO (Y_1),… , PRIMERO (Y_{i-1}); es decir, $Y_1\ldots Y_{i-1}$→ ϵ. Si ϵ esta en PRIMERO (Y_j) para toda j=1,2,…, k, entonces añádase ϵ a PRIMERO (X). Si Y_1

no deriva a ϵ, entonces no se añade nada mas a PRIMERO (X), pero si Y_1--> ϵ, entonces se le añade PRIMERO (Y_2), y así sucesivamente.

Ahora se puede calcular PRIMERO para cualquier cadena $X_1 X_2 \ldots X_n$ de la siguiente forma: añádanse a PRIMERO ($X_1X_2 \ldots X_n$). Todos los símbolos distintos de ϵ de PRIMERO (X_1). Si ϵ está en primero (X_1), añádanse también los símbolos distintos de ϵ de PRIMERO(X_2); si ϵ está tanto en PRIMERO(X_1) como en PRIMERO (X_2), añádanse también los símbolos distintos de ϵ de PRIMERO(X_3), y así sucesivamente por ultimo añádanse ϵ a PRIMERO($X_1X_2 \ldots X_n$) si, para toda i, PRIMERO(X_i) contiene ϵ.

Para calcular SIGUIENTE (A) para todo los no terminales A, aplíquense las reglas siguientes hasta que no se pueda añadir nada mas a ningún conjunto SIGUIENTE.

1. Póngase $ en SIGUIENTE(S), donde S es el símbolo inicial y $ es delimitador derecho de la entrada.
2. Si hay una producción A→ αBβ, entonces todo lo que este en PRIMERO (β) excepto ϵ se pone en SIGUIENTE (B).
3. Si hay una producción A→ αB o una producción A→αBβ, donde PRIMERO (β) contenga ϵ (es decir, β→ ϵ) , entonces todo lo que este en SIGUIENTE (A) se pone en SIGUIENTE (B).

Ejemplo 4.9. Considérese la siguiente gramática G9:

E→TE´

E´→+TE´| ϵ

T→FT´

T´→*FT´| ϵ

F→(E)|**id**

Entonces

PRIMERO(E)=PRIMERO(T)=PRIMERO(F)={(, **id**}

PRIMERO(E´)={+, **ϵ** }

SIGUIENTE(T´)=(*, **ϵ)**

SIGUIENTE(E)=SIGUIENTE(E´)={),$}

SIGUIENTE(T)=SIGUIENTE(T´)={+,),$}

SIGUIENTE(F)={+,*,),$}

Por ejemplo, se añaden **id** y el paréntesis izquierdo a PRIMERO(F) por la regla 3 de la definición de PRIMERO, con i=1 en cada caso, puesto que PRIMERO(**id**)=(**id**) y PRIMERO(´(´)={(} por la regla 1. Entonces por la regla 3, con i=1, la producción T→FT´ supone que **id** y el paréntesis izquierdo están asimismo en PRIMERO(T). Otro ejemplo más, **ϵ** esta en PRIMERO(E´) por la regla 2.

Para calcular los conjuntos SIGUIENTE, se pone $ en SIGUIENTE(E) por la regla 1 de SIGUIENTE. Por la regla 2 aplicada a la producción F→ (E), el paréntesis derecho también esta en derecho están en SIGUIENTE(E´). Como E´→ **ϵ**, también están en siguiente(T). Como último ejemplo de la aplicación de las reglas de SIGUIENTE, la producción E→TE´ supone, por la regla 2, que todo lo que este en PRIMERO(E´), salvo **ϵ,** debe ponerse en SIGUIENTE(T). Ya se ha visto que $ esta en SIGUIENTE(T).

Construcción de tablas de análisis sintáctico

Se puede utilizar el siguiente algoritmo para construir una tabla de análisis sintáctico predictivo para una gramática G. La idea en que se basa el algoritmo es la siguiente. Supóngase que A→α es una producción con a en PRIMERO(α). Entonces, el analizador sintáctico expandirá *A* por α cuando el símbolo actual de la entrada sea a. la única complicación surge cuando α =b **ϵ** o α→ . en este caso, se debe expandir de nuevo *A* en α si el símbolo actual de la entrada esta en SIGUIENTE(*A*), o si ya se ha alcanzado en $ de la entrada y $ esta en SIGUIENTE(*A*).

Algoritmo 4.4. Construcción de una tabla de análisis sintáctico predictivo.

Entrada. Una gramática G.

Salida. La tabla de análisis sintáctico M.

Método.

1. Para cada producción *A→α* de la gramática, dense los pasos 2 y 3.
2. Para cada terminal *a* de PRIMERO(α) , a;adase *A*→ α a M[*A.a*].
3. Si **ϵ** esta en PRIMERO(α), a;adase *A*→ α a M[*A,b*] para cada terminal b de SIGUIENTE(A). Si **ϵ** esta en PRIMERO(**α**) Y $ esta en SIGUIENTE(A), a;adase A→ **α** a M[A,$].
4. Hágase que cada entrada no defina de *M* sea **error.**

Una gramática cuya tabla de análisis sintáctico no tiene entradas con definiciones múltiples se define como LL(1). La primera "L" de LL(1) representa (por left, en ingles, izquierda) el examen de la entrada de izquierda a derecha, la segunda "L" representa una derivación por la izquierda, y el "1" es por utilizar un símbolo de entrada de examen por anticipado a cada paso para tomar las decisiones de la acción en el análisis sintáctico.

Las gramáticas LL(1) tienen varias propiedades distintas. Ninguna gramática ambigua o recursiva por la izquierda puede ser LL(1). También se puede demostrar que una gramática G es LL(1) si, y solo si, cuando A → α |β sean dos producciones distintas de G se cumplen las siguientes condiciones:

1. Para ningún terminal *a* tanto α como β derivan a la vez cadenas que comiencen con *a*.
2. A lo sumo una de α y β puede derivar la cadena vacía.
3. Si β →ϵ, α no deriva ninguna cadena que comience con un terminal en SIGUIENTE(A).

Recuperación de errores en el análisis sintáctico predictivo.

La pila de un analizador sintáctico no recursivo hace explicito los terminales y no terminales que analizador espera emparejar con el resto de la entrada por tanto se dará referencia a los símbolos de la pila de un analizador sintáctico en la siguiente exposición. Durante el análisis sintáctico predictivo se detecta un error cuando el terminal de la sima de la pila no concuerda con el siguiente símbolo de entrada o cuando el no terminal A esta en la cima de la pila, *a* es el siguiente símbolo de entrada, y la entrada M[A,*a*] de la tabla de análisis sintáctico esta vacía.

La recuperación en modo de pánico se basa en la idea de saltarse símbolos de la entrada hasta que aparezca un componente léxico que pertenezca a un conjunto seleccionado de componentes léxicos de sincronización. Su efectividad depende de la elección del conjunto de sincronización. Los conjuntos deben elegirse de forma que el analizador sintáctico se recupere con rapidez de los errores con más probabilidades de ocurrir en la práctica. Algunas técnicas heurísticas son las siguientes:

1. Como punto de partida, se pueden colocar todos los símbolos de SIGUIENTE (A) dentro del conjunto de sincronización para el no terminal A. Si se saltan componentes léxicos hasta encontrar un elemento de SIGUIENTE (A) y se saca a A de la fila, es probable que el análisis sintácticos pueda continuar.

2. No es suficiente usar SIGUIENTE (A) como conjunto de sincronizaciones para A. Por ejemplo, si los símbolos de punto y coma terminan las proposiciones, como en C, entonces las palabras claves que comienzan proporciones pueden no aparecer en el conjunto SIGUIENTE del no terminal que genera las expresiones. Por tanto, un punto y coma que falte después de una asignación puede dar como resultado que se salte la palabra clave que indique la siguiente proposición. A menudo hay una estructura jerárquica en las construcciones de un lenguaje; por ejemplo la expresiones aparecen dentro de proposiciones, las cuales aparecen dentro de bloques, y así sucesivamente. Se puede añadir al conjunto de sincronización de una construcción de menor jerarquía los símbolos que inician las construcciones de mayor jerarquía. Por ejemplo, se pueden agregar palabras claves que

comienzan proposiciones a los conjuntos de sincronización para los no terminales que generan expresiones.

3. Si se añaden símbolos de PRIMERO (A) al conjunto de sincronización para el no terminal A, entonces se puede continuar el análisis sintáctico según A si aparece en la entrada un símbolo de PRIMERO (A).

4. Si un no terminal puede generar la cadena vacía, se puede usar la producción que derive a ϵ como alternativa por omisión. Esto puede posponer alguna detección de errores pero no la omisión de error. Este método reduce el número de no terminales que hay que considerar durante la recuperación del error.

5. Si no se puede emparejar un terminal de la cima de la pila, una idea sencilla es sacar el terminal, emitir un mensaje que indique que se inserto el terminal y continuar el análisis. En realidad, este método considera al conjunto de sincronización de un componente léxico como si estuviera compuesto por todos los otros componentes léxicos.

La tabla que mostramos a continuación debe utilizarse de la forma siguiente. Si el analizador sintáctico busca la entrada M[A,*a*] y ve que esta en blanco, debe saltarse el símbolo de entrada *a*. Si la entrada es sinc, se saca el no terminal de la cima de la pila para continuar el análisis. Si un componente léxico de la sima de la pila no concuerda con el símbolo de entrada, entonces se saca el componente léxico de la pila, como ya se ha mencionado.

Con la entrada errónea)id*+id, el analizador sintáctico y el mecanismo de recuperación de errores de esta primer tabla se comportan como en la segunda tabla.

NO TERMINAL	**SIMBOLO DE ENTRADA**					
	Id	**+**	*****	**(**	**)**	**$**
E	E→TE'			E→TE'	Sinc	Sinc
E'		E'→+TE'			E'→ϵ	E'→ϵ
T	T→FT'	sinc		T→FT'	Sinc	Sinc
T'		T'→ϵ	T'→*FT'		T'→ϵ	T'→ϵ
F	F→id	sinc	sinc	F→(E)	Sinc	Sinc

Componentes léxicos de sincronización añadidos a la tabla de análisis sintáctico.

PILA	ENTRADA	COMENTARIO
$E	) id*+id$	Error, saltar)
$E	id*+id$	Id estaá en PRIMERO (E)
$E'T'	id*+id$	
$E'T'F	id*+id$	
$E'T'id	id*+id$	
$E'T'	*+id$	
$E'T' F*	*+id$	
$E'T' F	+id$	Error, M[F,+]=sinc
$E'T'	+id$	F ha sido extraída de la pila
$E'	+id$	
$E'T +	+id$	
$E'T	id$	
$E'T ' F	id$	
$E'T' id	id$	
$E'T'	$	
$E'	$	
$	$	

Fig. 4.3. Movimientos para el análisis y la recuperación de errores realizados por el analizador sintáctico predictivo.

Tema 5. TÉCNICA DE PARSEO LR Y LL

Objetivos

- Aplica técnica de Parseo LL y sus elementos.
- Identifica las técnicas usadas por el analizador Bison, para generar el parser.
- Aplica técnica de parseo LR.

Técnica de Parseo LL.

Gramáticas y Parsers Ll (1)

En temas anteriores aprendimos las herramientas del parseo descendente recursivo. Ahora estudiaremos la gramática LL(1), que son la clase de CFG`s adaptables para el parseo descendente recursivo. También definiremos los parsers LL(1), los cuales usan una tabla de parseo LL(1) en vez de procedimientos recursivos para controlar el parseo.

La función Predic de LL(1)

El problema principal construyendo un procedimiento de parseo es la decisión de que producción verificar (match). La decisión puede ser formalizada definiendo una función Predict que examina el símbolo lookahead para deducir la producción que debe ser usada para expandir cada no terminal. Considere la producción $A \rightarrow X_1 \ldots X_m$, $m \geq 0$. Necesitamos computar el conjunto de posibles lookahead (tokens) que pueden indicar que esta producción es la que se va a verificar. Este conjunto es claramente aquellos terminales que pueden ser producidos por $X_1 \ldots X_m$. Ya que el lookahead es un simple token, queremos el conjunto de primeros tokens que pueden ser producidos. A como vimos anteriormente este conjunto de tokens es el conjunto **Primero**($X_1 \ldots X_m$). Primero consideremos el símbolo de la izquierda X_1. Si este es un terminal, entonces Primero($X_1 \ldots X_m$) = X_1. Si X_1 no es terminal entonces computamos el conjunto Primero para cada lado derecho correspondiente a X_1. Qué pasa si X_1 genera λ? Entonces X_1 puede ser borrado y Primero($X_1 \ldots X_m$) depende de X_2. En particular, si X_2 es un terminal, este es incluido en Primero($X_1 \ldots X_m$). Si X_2 es un no terminal, computamos el conjunto Primero para cada uno de su correspondiente lado derecho. Similarmente, si ambos X_1 y X_2 producen λ, consideremos X_3 y así sucesivamente. Qué pasaría si el lado derecho entero produce λ? Entonces el lookahead será determinado por aquellos terminales que pueden seguir el lado izquierdo de la producción (A en nuestro ejemplo). Usamos entonces Siguiente(A) el cual es el conjunto de tokens que pueden seguir a A en alguna derivación legal.

Ahora definimos el conjunto de tokens lookahead que causaran la predicción de la producción $A \rightarrow X_1 \ldots X_m$. Llamamos a este conjunto Predict:

$$\text{Predict}(A \rightarrow X_1 \dots X_m) = \begin{cases} (\text{Primero}(X_1 \dots X_m) - \lambda) \cup \text{Siguiente}(A) \\ \quad \text{Si } \lambda \in \text{Primero}(X_1 \dots X_m) \\ \text{Primero}(X_1 \dots X_m) \\ \quad \text{Si c} \notin \text{Primero}(X_1 \dots X_m) \end{cases}$$

Esto es, cualquier token que puede ser el primer símbolo producido por el lado derecho de una producción predecirá esta producción. Además si todo el lado derecho puede producir λ [$\lambda \in$ Primero($X_1 \dots X_m$)] entonces tokens que siguen inmediatamente al lado izquierdo de una producción puede también predecir esta producción.

Porque λ no es un símbolo terminal, este no puede ser un lookahead y por lo tanto no puede ser incluido en ningún conjunto PREDICT.

Una ultimo consideración. Qué tal si para 2 producciones, $A \rightarrow X_1 \dots X_m$ y $A \rightarrow Y_1 \dots Y_p$ tenemos algún token *t* para el cual $t \in$ Predict($A \rightarrow X_1 \dots X_m$) y $t \in$ Predict($A \rightarrow Y_1 \dots Y_p$); esto es, el mismo token predice más de una producción?

Este conflicto excluirá esta gramática de la clase de gramáticas LL(1). LL(1) contiene exactamente aquellas gramáticas que tienen conjuntos Predict disjuntos para producciones que comparten un lado izquierdo común.

Por experiencia hemos aprendido que usualmente es posible crear una gramática LL(1) para un lenguaje de programación.

No todas las gramáticas son LL(1); sin embargo, muchas gramáticas que no son LL(1) pertenecen a otras, más complicadas, clases de gramáticas para las cuales sus parsers pueden ser construidos automáticamente.

CFG en forma estándar para MICRO

1. System_goal →program **SCANEOF**
2. Program →**BEGIN** st_list **END**
3. St_list →stat st_tail
4. St_tail →stat st_tail
5. St_tail →λ
6. Stat →**ID ASSIGNOP** expr **SEMICOLON**
7. Stat →**READ LPAREN** id_list **RPAREN SEMICOLON**
8. Stat →**WRITE LPAREN** expr_list **RPAREN SEMICOLON**
9. Id_list →ID id_tail
10. Id_tail →**COMMA ID** id_tail
11. Id_tail →λ
12. Expr_list →expr expr_tail
13. Expr_tail →**COMMA** expr expr_tail
14. Expr_tail → λ
15. Expr →prim prim_tail
16. Prim_tail →add_op prim_tail
17. Prim_tail → λ
18. Prim →**LPAREN** expr **RPAREN**
19. Prim →**ID**
20. Prim →**INT LITERAL**
21. Add_op →**PLUSOP**
22. Add_op →**MINUSOP**

Esta forma estándar se obtuvo haciendo uso del algoritmo descrito anteriormente.

Ejemplo 5.1. Calculo de los conjuntos: Primero, Siguiente y Predict para MICRO.

Conjuntos Primero para MICRO

No terminal	Conjuntos Primero
System_goal	{begin}
Program	{begin}
St_list	{ID,read,write}
Stat	{ID,read,write}
St_tail	{ID,read,write,λ}
Expr	{ID,INTLIT,(}
Id_list	{ID}
Expr_list	{ID,INTLIT,(}
Id_tail	{COMMA,λ}
Expr_tail	{COMMA,λ}
Prim	{ID,INTLIT,(}
Prim_tail	{+,-,λ}
Add_op	{+,-}

Fig. 5.1. Tabla de Conjuntos Primero para micro.

Conjuntos Siguiente para MICRO

No terminal	Conjuntos Siguiente
Program	{$}
St_list	{end}
Stat	{ID,read,write,end}
St_tail	{end}
Expr	{COMMA,SEMICOLON,)}
Id_list	{)}
Expr_list	{)}
Id_tail	{)}
Expr_tail	{)}
Prim	{COMMA,SEMICOLON,+,-,)}
Prim_tail	{COMA,SEMICOLON,)}
Add_op	{ID,INTLIT,(}
System_goal	{λ}

Fig. 5.2. Tabla de Conjuntos Siguiente para micro.

Conjuntos Predict para MICRO.

Prod	**Conjuntos Predict**		
1	Primero(program $)=	Primero(program)=	{begin}
2	Primero{begin st_list end}=	Primero(begin)=	{begin}
3	Primero(stat st_tail)=	Primero(stat)=	{ID,read,write}
4	Primero(stat st_tail)=	Primero(stat)=	{ID,read,write}
5	Primero(λ) υ Siguiente()st_tail)- λ =	Siguiente(st_tail)	{end}
6	Primero(ID:=expr;)=	Primero(ID)=	{ID}
7	Primero(read(id_list);)=	Primero(read)=	{read}
8	Primero(write(expr_list);)=	Primero(write)=	{write}
9	Primero(ID, id_tail)=	Primero(ID)=	{ID}
10	Primero(,ID id_tail)=	Primero(,)=	{,}
11	Primero(λ) υ Siguiente(id_tail)- λ =	Siguiente(id_tail)=	{)}
12	Primero(expr expr_tail)=	Primero(expr_tail)=	{ID,INTLIT,(}
13	Primero(,expr expr_tail)=	Primero(,)=	{,}
14	Primero(λ) υ Siguiente(expr_tail)- λ =	Siguiente(expr_tail)=	{)}
15	Primero(prim prim_tail)=	Primero(prim)	{ID,INTLIT,(}
16	Primero(add_op prim prim_tail)=	Primero(add_op)=	{+,-}
17	Primero(λ) υ Siguiente (expr_tail)- λ =	Siguiente(prim_tail)=	{COMMA,;,)}
18	Primero((expr))=	Primero(()=	{(}
19	Primero(ID)=		{ID}
20	Primero(INTLIT)=		{INTLIT}
21	Primero(+)=		{+}
22	Primero(-)=		{-}

Fig. 5.3. **Tabla de Conjuntos Predic para micro.**

Tabla de Parseo LL(1)

La información contenida en la función Predict puede ser convenientemente representada en una tabla de parseo LL(1). Esta tabla, T, es de la forma $T : V_n \times V_t \rightarrow P \cup \{Error\}$ donde P es el conjunto de todas las producciones. Si A es un no terminal a ser verificado y T es el token lookahead, T[A][t] nos dice que producción predecir. Si no hay una producción apropiada, T[A][t] produce un Error indicando error de sintaxis. T es definida así:

$$T[A][t] = \begin{cases} A \rightarrow X_1 \ldots X_m, \text{ si } t \in Predict(A \rightarrow X_1 \ldots X_m) \\ Error, \text{ en otros casos} \end{cases}$$

Una gramática G es LL(1) si todas las entradas en T contienen una única predicción o una señal de Error. De otra manera, G es LL(1) si para cualquier no terminal A y para cualquier símbolo y $A \rightarrow \beta$ son producciones distintas.

Tabla LL(1) para Micro

	ID	**INTLIT**	**:=**	**,**	**;**	**=**	**-**	**(**	**)**	**begin**	**end**	**read**	**write**	**$**
System_goal										1				
Program										2				
St_list	3											3	3	
Stat	6											7	8	
St_tail	4										5	4	4	
Expr	15	15							15					
Id_list	9													
Expr_list	12	12							12					
Id_tail				10					11					
Expr_tail				13					14					
Prim	19	20						18						
Prim_tail				17	17	16	16		17					
Add_op														

Fig. 5.4. Tabla LL(1) para Micro.

Construyendo parsers recursivamente descendente a partir de tablas LL(1).

Los procedimientos de parseo son de la forma

```
void no_term(void)
{
    token tok=next_token();
    swith(tok){
    case TERMINAL – LIST;
        parsing_actions();
        break;
        …
    default;
        syntax_error(tok);
        break;
   }
}
```

Ahora consideremos un algoritmo que automáticamente crea procedimientos de parseo a como los del ejemplo anterior a partir de la tabla LL(1) de una gramática. Los valores de TERMINAL_LIST y parsing_actions() serán determinados por la tabla LL(1).

Ahora aumentamos la estructura de dato para describir una gramática con el nombre de los símbolos en la gramática. Una gramática es ahora

```
  typedef int symbol;
#define VOCABULARY (NUM_TERMINALS + NUM_NONTERMINALS)
typedef struct gram{
  ...
  ...
  }productions [NUM_PRODUCTIONS];
   symbol vocabulary [VOCABULARY];
   char *names[VOCABULARY];
  }grammar;

typedef struct prod production;
typedef symbol terminal;
typedef symbol nonterminal;
```

Asumamos que tenemos un arreglo de símbolos de una gramática para verificar en un procedimiento de parseo. La rutina gen_actions() toma los símbolos y genera las acciones (llama los procedimientos de parseo y a match()) necesarias para verificarlos en un parser recursivo descendente.

gen_actions() asume la función make_id() que toma el nombre de un símbolo y transforma en un identificador de programa valido.

Ej.: make_id("st_list")
devuelve "statementlist"
make_id(":=")
devuelve "ASSIGNOP"

making_parsing_proc() toma un no terminal y una tabla LL(1) de MICRO.

Algoritmo para generar acciones recursivas descendentes.

```
extern char *make-id(char*);

void gen_actions(symbol x[],int x_length)
{
  int i;
  char *id;

/*Generate recursive descent actions needed to match x. */
   if(x_length==0)
       printf("; /*null*/\n");
     else{
          for(i=0;i<x_length;i++){
             id=make_id(g.names[x[i]]);
             if(is_terminal(x[i]))
               printf("\t\t match(%s);\n", id);
             else
               printf("\t\t %s(); \n",id);
     }
  }
}
```

Algoritmo para generar procedimientos de parseo

```
void make_parsing_proc(const nonterminal A, const Itable T)
{
/*  Generate recursive descent parsing procedure for A.*/
      extern grammar g;
      terminal x;
      int i,j;

      printf("void %s(void)\n{\n", make_id(g.names[A]))
      printf("\t token tok=net_token()\n");
      printf("\t switch(tok){\n");

   /*for each production where A is the LHS */
     for(i=0;i<g.num_productions;i++){
       if(g.productions[i].lhs !=A)
            continue;

        p=g.productions[i];
     /*for each terminal in the grammar*/
       for(j=0;j<NUM_TERMINALS;j++){
             x=g.terminals[j];
             if(T[A][x]==i)/* this production*/
               printf("\tcase %s:\n", make_id(g.names[x]));
           }
           gen_actions(p.rhs,p.rhs_length);
           printf("\t\tbreak;\n");
     }
     printf("\tdefault:\n");
     printf("\tsyntax_error(tok);\n");
     printf("\tbreak;\n\t}\n");
}
```

Un conductor para Parser LL(1)

Los procedimientos de parseo recursivo descendente son normalmente aumentados con código que preforma análisis semántico y generación de código. A causa de esto, los procedimientos de parseo, una vez integrados en un compilador, no son fáciles de cambiar. Lo deseable es tener una vía de actualizar un parser sin afectar innecesariamente otros componentes del compilador.

En vez de usar una tabla LL(1) para construir procedimientos de parseo, es posible usar la tabla en conjunción con un programa conductor (driver) para formar un parser LL(1).

Las tablas LL(1) son computadas solo una vez cuando el parser es construido. Las tablas son consideradas solo de lectura por el conductor LL(1), cambiar la gramática y construir un parser un parser nuevo es fácil nuevas tablas LL(1) son computadas y substituyen a las anteriores.

Ya que el conductor LL(1) usa una pila (stack) en vez de llamadas a procedimientos recursivos para almacenar símbolos aun a ser verificados, el parser resultante puede ser esperado sea más pequeño y más rápido que su correspondiente parser recursivo descendente.

Un conductor para Parser LL(1)

```
void lldriver(void)
{
  /*Push the start symbol onto an empty stack. */
      push(s);
      while(! stack_empty()){
          /*let X be the top stack symbol; */
          /*let a be the current input token. */
       if(is_nonterminal(X)&& T[X][a]==X--> Y1...Ym){
       /*Expand non-terminal */
          pop(1);
          push Ym, Ym-1, ... Y1 onto the stack;
        }else if(X==a){                  /*X in terminal*/
            pop(1);                      /*Match of X worked*/
            scanner(&a);                 /*Get next token*/
         }else
             /*Process syntax error.*/
         }
}
```

Este conductor LL(1) es muy sencillo. Este empuja (stacks) símbolos que serán verificados o expandidos. Símbolos terminales en la pila deben verificarse(match) con el símbolo de entradas; símbolos de entrada; símbolos no terminales son expandidos usando la tabla LL(1).

Parseo LL(1) para: begin A:= BB – 314 + A; end $

Step	Parser Action	Remainig Input	Parse Stack
(1)	Predict 1	Begin A:=BB-314+A;end$	System_goal
(2)	Predict 2	Begin A:=BB-314+A;end$	Program$
(3)	Match	Begin A:=BB-314+A;end$	Begin st_list end $
(4)	Predict 3	A:=BB-314+A;end$	St_list end $
(5)	Predict 6	A:=BB-314+A;end$	Stat st_tail end $
(6)	Match	A:=BB-314+A;end$	ID := expr; st_tail end $
(7)	Match	:=BB-314+A;end$	:=expr;st_tail end $
(8)	Predict 15	BB-314+A;end$	Expr;st_tail end $
(9)	Predict 19	BB-314+A;end$	Prim prim_tail;st_tail end $
(10)	Match	BB-314+A;end$	ID prim_tail;st_tail end $
(11)	Predict 16	-314+A;end$	Prim_tail;st_tail end $
(12)	Predict 22	-314+A;end$	Add_op prim prim_tail;st_tail end $
(13)	Match	-314+A;end$	- prim prim_tail;st_tail end $
(14)	Predict 20	314+A;end$	Prim prim_tail;st_tail end $
(15)	Match	314+A;end$	Intliteral prim_tail;st_tail end $
(16)	Predict 16	+A;end$	Prim_tail; st_tail end $
(17)	Predict 21	+A;end$	Add_op prim prim_tail;st_tail end $
(18)	Match	+A;end$	+ prim prim_tail;st_tail end $
(19)	Predict 19	A;end$	Prim prim_tail;st_tail end $
(20)	Match	A;end$	ID prim_tail; st_tail end $
(21)	Predict 17	;end$	Prim_tail;st_tail end $
(22)	Match	;end$	;st_tail end $
(23)	Predict5	end$	St_tail end $
(24)	Match	end$	End $
(25)	Macth	$	$

Fig. 5.5. Tabla de Parseo LL(1) para: begin A:= BB – 314 + A; end $.

Un conductor para Parser LL(1)

En vez de usar tabla LL(1) para construir procedimientos de parseo, es posible usar la tabla en conjunción con un programa conductor (driver) para formar un parser LL(1).

Las tablas LL(1) son computadas solo una vez cuando el parser es construido. Las tablas son consideradas solo de lectura por el conductor LL(1) que las usa para controlar el parseo. Ya que el mismo conductor es usado con todas las tablas LL(1), cambiar la gramática y construir un parser nuevo es fácil nuevas tablas LL(1) son computadas y substituyen a las anteriores.

Ya que el conductor LL(1) usa una pila (stack) en vez de llamadas a procedimientos recursivos para almacenar símbolos aun a ser verificados, el parser resultante puede ser esperado sea más pequeño y más rápido que su correspondiente parser recursivo descendente.

Haciendo Gramáticas LL(1)

No siempre es fácil para escritores de compiladores crear gramáticas LL(1). El problema es que LL(1) requiere una única predicción para cada combinación de no terminal y cada símbolo lookahead, y no es difícil escribir producciones que violen el requerimiento de predicción única.

Afortunadamente, la mayoría de los conflictos de predicción LL(1) pueden ser agrupados en dos categorías: prefijos comunes y recursión izquierda. algunas transformaciones gramaticales que eliminan prefijos comunes y recursión izquierda son conocidas, y esas transformaciones nos permiten reescribir la mayoría de las formas LL(1) invalidas.

Categoría de conflictos prefijos comunes: dos producciones con el mismo lado izquierdo que tienen lados derechos que comparten un prefijo común.

Ejemplo:

stmt→ if expr then stmt_list endif;
stmt→ if expr then stmt_list else stmt_list endif;

El conflicto se produce ya que los conjuntos primero para cada lado derecho no serán disjuntos (al menos que el prefijo común genere solamente λ). La solución a conflictos de predicción causados por prefijos comunes es una simple transformación de factorización a como se muestra a continuación.

void factor (grammar *G)
{
while(G tiene dos o más producciones con el mismo LHS y un prefijo común)\
{
Let S={*A*-->αβ,..., *A*-->α Γ} el conjunto de producciones con el mismo lado izquierdo, *A*, y el prefijo común α

Crear un nuevo no terminal, *N*;

Reemplazar *S* con el conjunto de producciones
    SET_OF(A-->α*N*,*N*-->β, *N*-->Γ)
}
}

Usando nuestro if-then-else anterior, factor() produce

stmt --> if expr then stmt_list if_suffix
if_suffix --> endif;
if_suffix--> else stmt_list endif;

En la segunda categoría de conflictos, una producción es recursiva por la izquierda si el símbolo al lado izquierdo es también el primer símbolo a su lado derecho. Por ejemplo, la producción E-->E+T es recursiva por la izquierda.

Gramáticas con producciones recursivas por la izquierda nunca pueden ser LL(1).

El algoritmo remove_left_recursion() remueve recursiones por la izquierda de una gramática factorizada.

Bison

Bison es un generador de analizadores sintácticos de propósito general que convierte una descripción gramatical para una gramática independiente del contexto LALR(1) en un programa en C que analice esa gramática. Una vez que sea un experimentado en Bison, podrá utilizarlo para desarrollar un amplio rango de analizadores de lenguajes, desde aquellos usados en simples calculadoras de escritorio hasta complejos lenguajes de programación.

Lenguajes y Gramáticas independientes del Contexto

Para que Bison analice un lenguaje, este debe ser descrito por una gramática independiente del contexto. Esto quiere decir que debe especificar uno o más grupos sintácticos y dar reglas para construirlos desde sus partes. Por ejemplo, en el lenguaje C, un tipo de agrupación son las llamadas `expresiones'.

Cualquier gramática expresada en BNF es una gramática independiente del contexto. La entrada de Bison es en esencia una BNF legible por la máquina.

No todos los lenguajes independientes del contexto pueden ser manejados por Bison, únicamente aquellos que sean LALR(1). Brevemente, esto quiere decir que debe ser posible decir como analizar cualquier porción de una cadena de entrada con un solo token de preanalisis. Hablando estrictamente, esto es una descripción de una gramática LR(1), y la LALR(1) implica restricciones adicionales que son difíciles de explicar de manera sencilla; pero es raro en la práctica real que se encuentre una gramática LR(1) que no sea LALR(1).

Denominamos token a un fragmento de la entrada que corresponde a un solo símbolo terminal, y grupo a un fragmento que corresponde a un solo símbolo no terminal.

Podemos utilizar el lenguaje C como ejemplo de que significan los símbolos, terminales y no terminales. Los tokens de C son los identificadores, constantes (numéricas y cadenas de caracteres), y las diversas palabras reservadas, operadores aritméticos y marcas de puntuación. Luego los símbolos terminales de una gramática para C incluyen `identificador', `numero', `cadena de caracteres', más un símbolo para cada palabra reservada, operador o marca de puntuación: `if', `return', `const', `static', `int', `char', `signo-mas', `llave-abrir', `llave-cerrar', `coma' y muchos más. (Estos tokens se pueden subdividir en caracteres, pero eso es una cuestión léxica, no gramatical.)

Aquí hay una función simple en C subdividida en tokens:

```
int  /* palabra reservada `int' */
cuadrado (x)  /* identificador, paréntesis-abrir */
/* identificador, paréntesis-cerrar */
int x;  /* palabra reservada `int', identificador, punto y coma */
{  /* llave-abrir */
return x * x;  /* palabra reservada `return', identificador, */
/* asterisco, identificador, punto y coma */
} /* llave-cerrar */
```

Las agrupaciones sintácticas de C incluyen a las expresiones, las sentencias, las declaraciones, y las definiciones de funciones. Estas se representan en la gramática de C por los símbolos no terminales `expresión', `sentencia', `declaración' y `definición de función'. La gramática completa utiliza docenas de construcciones del lenguaje adicionales, cada uno con su propio símbolo no terminal, de manera que exprese el significado de esos cuatro. El ejemplo anterior es la definición de una función; contiene una declaración, y una sentencia. En la sentencia, cada `x' es una expresión y también lo es `x * x'. Cada s no terminal debe poseer reglas gramaticales mostrando como está compuesto de construcciones más simples. Por ejemplo, un tipo de sentencia en C es la sentencia return; esta será descrita con una regla gramatical que interpretada informalmente será así:

Una `sentencia' puede estar compuesta de una palabra clave `return', una `expresión' y un `punto y coma'.

El analizador de Bison lee una secuencia de tokens como entrada, y agrupa los tokens utilizando las reglas gramaticales. Si la entrada es válida, el resultado final es que la secuencia de tokens entera se reduce a una sola agrupación cuyo símbolo es el símbolo de arranque de la gramática. Si usamos una gramática para C, la entrada completa debe ser una `secuencia de definiciones y declaraciones'. Sino, el analizador informa de un error de sintaxis.

De las reglas formales a la entrada de Bison

Una gramática formal es una construcción matemática. Para definir el lenguaje para Bison, debe escribir un archivo expresando la gramática con la sintaxis de Bison: un archivo de gramática de Bison. Un símbolo no terminal en la gramática formal se representa en la entrada de Bison como un identificador, similar a un identificador en C. Por convención, deberán estar en minúsculas, tales como expr, stmt o declaración.

La representación en Bison para un símbolo terminal se llama también un tipo de token. Los tipos de tokens también se pueden representar como identificadores al estilo de C. Por convención, estos identificadores deberán estar en mayúsculas para distinguirlos de los no terminales: por ejemplo, INTEGER, IDENTIFICADOR, IF o RETURN. Un símbolo terminal que represente una palabra clave en particular en el lenguaje deberá bautizarse con el nombre después de pasarlo a mayúsculas.

El símbolo terminal error se reserva para la recuperación de errores. Un símbolo terminal puede representarse también como un carácter literal, al igual que una constante de carácter en C. Deberá hacer esto siempre que un token sea simplemente un único carácter (paréntesis, signo-mas, etc.): use el mismo carácter en un literal que el símbolo terminal para ese token.

Una tercera forma de representar un símbolo terminal es con una cadena de caracteres de C conteniendo varios caracteres. Las reglas gramaticales tienen también una expresión en la sintaxis de Bison. Por ejemplo, aquí esta la regla en Bison para una sentencia return de C. El punto y coma entre comillas es un token de carácter literal, representando parte de la sintaxis de C para la sentencia; el punto y coma al descubierto, y los dos puntos, es puntuación de Bison que se usa en todas las reglas.

```
stmt: RETURN expr ';'
;
```

Valores Semánticos

Una gramática formal selecciona tokens únicamente por sus clasificaciones: por ejemplo, si una regla menciona el símbolo terminal `constante entera', quiere decir que cualquier constante entera es gramaticalmente valida en esa posición. El valor preciso de la constante es irrelevante en cómo se analiza la entrada: si `x+4' es gramatical entonces `x+1' o `x+3989' es igualmente gramatical.
Pero el valor preciso es muy importante para lo que significa la entrada una vez que es analizada.

Un compilador es inservible si no puede distinguir entre 4, 1 y 3989 como constantes en el programa. Por lo tanto, cada token en una gramática de Bison tiene ambos, un tipo de token y un valor semántico. El tipo de token es un símbolo terminal definido en la gramática, tal como INTEGER, IDENTIFICADOR o ','. Este dice todo lo que se necesita para saber decidir donde podrá aparecer válidamente el token y como agruparlo con los otros tokens. Las reglas gramaticales no saben nada acerca de los tokens excepto de sus tipos.

Acciones Semánticas

Para que sea útil, un programa debe hacer algo más que analizar la entrada; este debe producir también alguna salida basada en la entrada. En una gramática de Bison, una regla gramatical puede tener una acción compuesta de sentencias en C. Cada vez que el analizador reconozca una correspondencia para esa regla, se ejecuta la acción. La mayor parte del tiempo, el propósito de una acción es computar el valor semántico de la construcción completa a partir de los valores semánticos de sus partes. Por ejemplo, suponga que tenemos una regla que dice que una expresión puede ser la suma de dos expresiones. Cuando el analizador reconozca tal suma, cada una de las subexpresiones posee un valor semantico que describe como fueron elaboradas. La acción para esta regla deberá crear un tipo de valor similar para la expresión mayor que se acaba de reconocer.

Por ejemplo, he aquí una regla que dice que una expresión puede ser la suma de dos subexpresiones:

```
expr: expr '+' expr { $$ = $1 + $3; };
```

La acción dice como producir el valor semántico de la expresión suma a partir de los valores de las dos subexpresiones.

La Salida de Bison: el Archivo del Analizador

Cuando ejecuta Bison, usted le da un archivo de gramática de Bison como entrada. La salida es un programa fuente en C que analiza el lenguaje descrito por la gramática. Este archivo se denomina un analizador de Bison. El trabajo del analizador de Bison es juntar tokens en agrupaciones de acuerdo a las reglas gramaticales por ejemplo, construir expresiones con identificadores y operadores. A medida que lo hace, este ejecuta las acciones de las reglas gramaticales que utiliza.

Los tokens provienen de una función llamada el analizador léxico que usted debe proveer de alguna manera (por ejemplo escribiéndola en C). El analizador de Bison llama al analizador léxico cada vez que quiera un nuevo token.

Etapas en el uso de Bison

El proceso real de diseño de lenguajes utilizando Bison, desde la especificación de la gramática hasta llegar a un compilador o intérprete funcional, se compone de estas etapas:

1. Especificar formalmente la gramática en un formato que reconozca Bison.
Para cada regla gramatical en el lenguaje, describir la acción que se va a tomar cuando una instancia de esa regla sea reconocida. La acción se describe por una secuencia de sentencias en C.

2. Escribir un analizador léxico para procesar la entrada y pasar tokens al analizador sintáctico.

Para hacer que este código fuente escrito se convierta en un programa ejecutable, debe seguir estos pasos:

1. Ejecutar Bison sobre la gramática para producir el analizador.
2. Compilar el código de salida de Bison, al igual que cualquier otro fichero fuente.
3. Enlazar los ficheros objeto para producir el producto final.

Gramáticas Independientes del Contexto

Muchas construcciones de los lenguajes de programación tienen una estructura inherentemente recursiva que se puede definir mediante gramáticas independientes del contexto. Por ejemplo, se puede tener una proposición condicional definida por una regla como Si S_1 y S_2 son proposiciones y E es una expresión, entonces
"if E then S_1 else S_2" es una proposición.

Análisis Sintáctico Descendente

En esta sección se introducen las ideas básicas del análisis sintáctico descendente y se enseña a construir una forma eficiente sin retroceso de un analizador sintáctico descendente llamada analizador sintáctico predictivo. Se define la clase de gramáticas LL(1), a partir de las cuales se pueden construir de manera automática analizadores sintácticos predictivos.

Análisis Sintáctico Por Descenso Recursivo

Se puede considerar el análisis sintáctico descendente como un intento de encontrar una derivación por la izquierda para una cadena de entrada. También se puede considerar como un intento de construir un árbol de análisis sintáctico para la entrada comenzando desde la raíz y creando los nodos del árbol en orden previo.

Analizadores Sintácticos Predictivos

En muchos casos, escribiendo con cuidado una gramática, eliminando su recursión por la izquierda y factorizando por la izquierda la gramática resultante, se puede obtener una gramática analizable con un analizador sintáctico por descenso recursivo que no necesita retroceso, es decir un analizador sintáctico predictivo. Para construir un analizador sintáctico predictivo, se debe conocer, dado el símbolo actual *a* de entrada y el no terminal *A* a expandir, cuál de las alternativas de producción $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ es la única alternativa que da lugar a una cadena que comience con *a*. Es decir, la alternativa apropiada debe ser detectable con solo ver el primer símbolo al que da lugar. Así se detectan generalmente las construcciones de flujo de control de la mayoría de lenguajes de programación, con sus palabras claves diferenciadoras. Por ejemplo, si tienen la producciones

Prop → **if** expr then prop **else** prop
| **while** expr **do** prop
| **begin** lista_props **end**

las palabras claves if, while y begin indican que alternativa es la única con posibilidad de éxito para encontrar una proposición.

En seguida se evidencian varias diferencias entre los diagramas de transiciones para un analizador léxico y para un analizador sintáctico predictivo. En el caso de un analizador sintáctico, hay un diagrama para cada no terminal. Las etiquetas de las aristas son componentes léxicos y no terminales. Una transición con un componente léxico (terminal) supone que se debe tomar dicha transición si ese componente léxico es el siguiente símbolo de entrada. Una transición con un no terminal A es una llamada al procedimiento para A.

Para construir el diagrama de transiciones de un analizador sintáctico predictivo a partir de una gramática, primero se debe eliminar la recursión por la izquierda de la gramática, y después factorizar dicha gramática por la izquierda. luego, para cada no terminal A se hace lo siguiente:

1. Créese un estado inicial y un estado final (de retorno).
2. Para cada producción $A \rightarrow X_1X_2\ldots$ Xn, créese un camino desde el estado inicial hasta el estado final, con aristas etiquetadas con $X1, X_2\ldots, X_n$.

El analizador sintáctico predictivo que se desprende de los diagramas de transiciones se comporta como sigue. Comienza en el estado de inicio del símbolo inicial. Si después de algunos movimientos se encuentran en el estado S con una arista etiquetada con el terminal a al estado t, y si el siguiente símbolo de entrada es a, entonces el analizador sintáctico cambia el cursor de la entrada una posición a la derecha y se va al estado t. si, por otro parte la arista esta etiquetada con un no terminal A, el analizador sintáctico va al estado de inicio de A, sin mover el cursor de la entrada. Si llega alcanzar el estado final de A, inmediatamente va al estado t, habiendo en efecto "leído" A de la entrada cuando se traslado del estado S al t. por último si hay una arista de S a t etiquetada con ϵ, el analizador sintáctico va inmediatamente del estado S al t sin avanzar la entrada.

Un programa para hacer un análisis sintáctico predictivo basado en un diagrama de transiciones intenta emparejar símbolos terminales con la entrada, y realiza una llamada potencialmente recursiva a un procedimiento siempre que deba seguir una arista etiquetada con un no terminal. Se puede obtener una implantación no recursiva con una pila para guardar los estados S cuando hay una transición con un no terminal saliendo de ese, y eliminado la pila al alcanzar el estado final de un no terminal. Muy pronto se analizara más detallada la implantación de diagramas de transición.

El enfoque anterior funciona si el diagrama de transiciones dado no presenta indeterminismo, en el sentido de que haya más de una transición desde un estado con la misma entrada. Si existe ambigüedad, se puede solucionar de una forma específica como se verá en el siguiente ejemplo. Si no se puede eliminar el indeterminismo, no se puede construir un analizador sintáctico predictivo, pero si un analizador sintáctico por descenso recursivo utilizando el retroceso para intentar sistemáticamente todas las posibilidades, si esa fuera la mejor estrategia de análisis posible.

Ejemplo 5.2. La figura 5.6. contiene un conjunto de diagramas de transiciones para la gramatica (G9). Las únicas ambigüedades se refieren a si se debe o no tomar una arista **є**. Si se interpreta que las aristas que salen del estado inicial de E' como indicativas de tomar la transición con + siempre que esta sea la entrada siguiente o tomar la transición con є en otro caso, y realizar el mismo supuesto para T', entonces se elimina la ambigüedad y se puede escribir un programa de análisis sintáctico predictivo para la gramatica (G9).

Se pueden simplificar los diagramas de transiciones sustituyendo unos diagramas por otros. Por ejemplo, en la figura 5.7(a), la llamada de E' se ha reemplazado a sí misma por un salto hasta el principio del diagrama de E'.

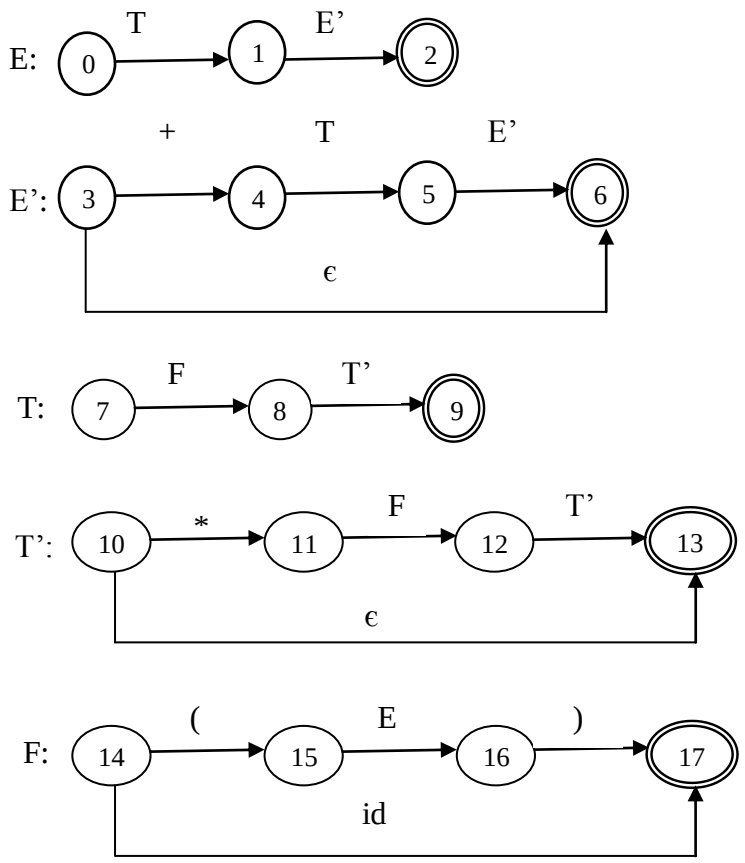


Fig. 5.6. Diagramas de transiciones para la gramatica (G9).

En la figura siguiente del inciso (b) se muestra un diagrama de transiciones equivalente para E'. Después se podría sustituir el diagrama de la figura de los diagramas de transiciones simplificados para la transición para E' en el diagrama de E de la figura de los diagramas de transiciones para la gramática (G9), obteniéndose el diagrama de la figura diagramas de transiciones simplificados inciso (c). por último se observa que el primero y tercer nodos de la figura diagramas de transiciones simplificados inciso (c) son equivalentes y se fusionan. El resultado, figura diagrama de transiciones simplificados inciso (d), se repite en el primer diagrama

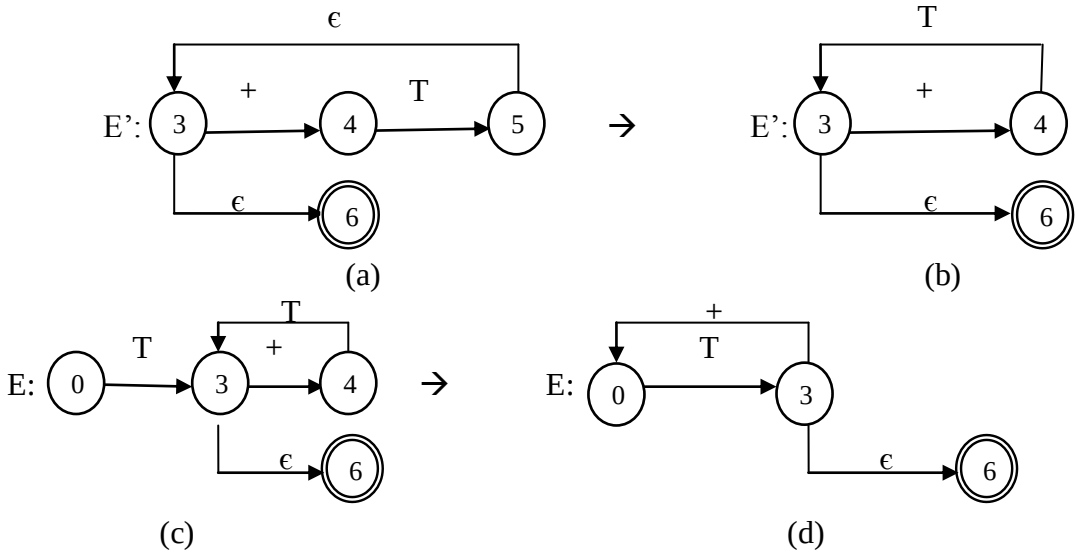
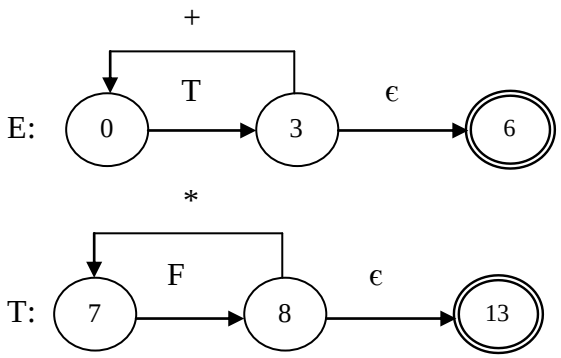


Fig. 5.7. Diagramas de transiciones simplificados.

de la figura diagramas de transiciones simplificados para las expresiones aritméticas. Las mismas técnicas sirven para los diagramas de T y T'. en la figura diagramas de transiciones simplificados para las expresiones aritméticas se muestra el conjunto completo de los diagramas obtenidos. Una implantación en C de este analizador sintáctico predictivo funciona un 20 ó 25 por 100 más rápidamente que una implantación en el mismo lenguaje de la figura 5.6.



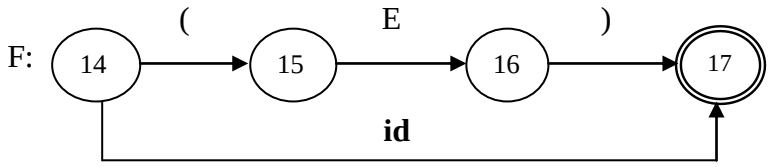


Fig. 5.8. Diagramas de transiciones simplificados para las expresiones aritméticas.

Análisis sintáctico predictivo no recursivo

Se puede construir un analizador sintáctico predictivo no recursivo explícitamente manteniendo una pila, en lugar de hacerlo implícitamente mediante llamadas recursivas. El problema clave durante el análisis sintáctico predictivo es determinar la producción que debe aplicarse a un no terminal. El analizador sintáctico no recursivo de la figura Modelo de un analizador sintáctico predictivo no recursivo busca la producción que debe aplicarse en una tabla de análisis sintáctico. A continuación se verá como se puede construir directamente la tabla a partir de ciertas gramáticas.

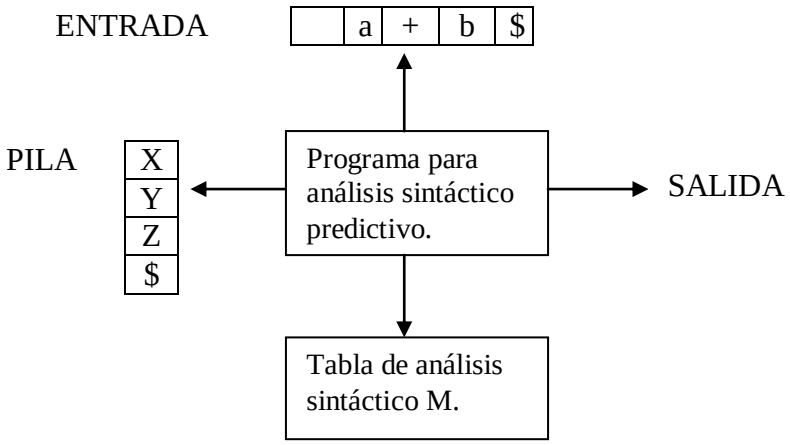


Figura 5.9. Modelo de un analizador sintáctico predictivo no recursivo.

Un analizador sintáctico predictivo guiado por tablas tiene un *buffer* de entrada, una pila, una tabla de análisis sintáctico y una cadena de salida. El *buffer* de entrada contiene la cadena que se va a analizar, seguida de $, un símbolo utilizado como delimitador derecho para indicar el fin de la cadena de entrada. La pila contiene una secuencia de símbolos gramaticales con $ en la parte de abajo, que indica la base de la pila. Al principio, la pila contiene el símbolo inicial de la gramática encima de $. La tabla de análisis sintáctico es una matriz bidimensional M[A, a], donde A es un no terminal, y a es un terminal o el símbolo $.

1. Si X=a=$, el analizador sintáctico se detiene y anuncia el éxito de la realización del análisis.
2. Si X=ą □ $, el analizador sintáctico saca a X de la pila y mueve el apuntador de entrada al siguiente símbolo de entrada.
3. Si X es un no terminal, el programa consulta la entrada M [X, a] de la tabla M de análisis sintáctico. Esta entrada será o una producción de X de la gramática o una entrada de error. Si, por ejemplo, M[X,a]={X→UVW}, el analizador sintáctico sustituye la X de la cima de la pila por WVU (con U en la cima). Como salida, se sabe que el analizador sintáctico solo imprime la producción utilizada; ahí se podría ejecutar cualquier otro código. Si M[X,a]= **error**, el analizador sintáctico llama a una rutina de recuperación de error.

Se puede describir el comportamiento del analizador sintáctico en función de sus configuraciones, que dan el contenido de la pila y la entrada restante.

Algoritmo 5.1. Análisis Sintáctico Predictivo No Recursivo.

Entrada. Una cadena w y una tabla de análisis sintáctico M para la gramática G.

Salida. Si w esta en L(G), una derivación por la izquierda de w; de lo contrario una indicación de error.

Método. Al principio, el analizador sintáctico esta en una configuración en la que tiene a $*S* en la pila con *S*, el símbolo inicial de G en el tope, y w$ en el *buffer* de entrada. En el siguiente fragmento de código se muestra el programa que utiliza la tabla de análisis sintáctico predictivo M para producir un análisis de la entrada.

Programa para análisis sintáctico predictivo.

```
 apuntar al primer símbolo de w$;
repeat
       sea X el símbolo de la cima de la pila y al símbolo apuntado por ae;
       if  X es un terminal o $ then
           if X= a then
             extraer X de la pila y avanzar ae
          else error()
 else  /* X es un no terminal */
         if M[X,a]=X→Y1Y2…Yk then begin
            extraer X de la pila;
            meter Yk, Yk-1,…, Y1 en la pila, con Y1 en la cima;
           emitir la producción X→Y1Y2…Yk
    end
    else error()
until X=$ /* la pila esta vacía*/
```

Considérese la gramática (G9) del ejemplo 4.8 en la figura 5.10 se muestra una tabla de análisis sintáctico predictivo para esta gramática. Los espacios en blanco son entradas de error; los otros espacios indican una producción con la cual expandir el no terminal de la cima en la pila. Obsérvese que aun no se ha indicado como seleccionar dichas entradas, pero se indicara en breve.

Con la entrada **id+id*id** el analizador sintáctico predictivo realiza la secuencia de movimiento de la figura **Movimiento realizado por el analizador sintáctico predictivo con la entrada id+id*id.** El apuntador de entrada apunta al símbolo de la extrema izquierda de la cadena en la columna ENTRADA. Si se observa con atención la opción de este analizador sintáctico, se nota que esta buscando una derivación por la izquierda para la entrada, es decir, las producciones emitidas son las de una derivación por la izquierda. los símbolos de entrada que ya se han examinados seguidos de los símbolos gramaticales de la pila (de la cima al fondo), son las formas de frase izquierdas de la derivación.

NO TERMINAL	SIMBOLO DE ENTRADA					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→**ϵ**	E'→**ϵ**
T	T→FT'			T→FT'		
T'		T'→**ϵ**	T→*FT'		T'→**ϵ**	T'→**ϵ**
F	F→**id**			F→(E)		

Fig. 5.10. Tabla de Análisis sintáctico M para la gramática (G9).

PILA	ENTRADA	SALIDA
$E	**id** + **id** * **id**$	
$E'T	**id** + **id** * **id**$	E→TE'
$E'T'F	**id** + **id** * **id**$	T→FT'
$E'T'**id**	**id** + **id** * **id**$	F→**id**
$E'T'	+ **id** * **id**$	
$E'	+ **id** * **id**$	T'→**ϵ**
$E'T+	+ **id** * **id**$	E→+TE'
$E'T	**id** * **id**$	
$E'T'F	**id** * **id**$	T→FT'
$E'T'**id**	**id** * **id**$	F→**id**
$E'T'	* **id**$	
$E'T'F*	* **id**$	T'→*FT'
$E'T'F	**id**$	
$E'T'**id**	**id**$	F→**id**
$E'T'	$	
$E'	$	T'→**ϵ**
$	$	E'→**ϵ**

Fig. 5.11. Movimiento realizado por el analizador sintáctico predictivo con la entrada **id+id*id.**

Analizadores Sintácticos LR.

En esta sección se analiza una técnica eficiente de análisis sintáctico ascendente que se puede utilizar para analizar una clase más amplia de gramáticas independientes del contexto. La técnica se denomina análisis sintáctico LR(k); la "L" es por el examen de la entrada de izquierda a derecha (en ingles, left-to-right), la "R" por construir una derivación por la derecha (en ingles, rightmost derivation) en orden inverso, y la k por el numero de símbolos de entrada de examen por anticipado utilizados para tomar las decisiones del análisis sintáctico. Cuando se omite, se asume que k, es 1. El análisis sintáctico LR es atractivo por varias razones.

- Se pueden construir analizadores sintácticos LR para reconocer prácticamente todas las construcciones de los lenguajes de programación para los que se pueden escribir gramáticas independientes del contexto.
- El método de análisis sintáctico LR es el método de análisis por desplazamiento y reducción sin retroceso más general que se conoce, y sin embargo se puede aplicar tan eficientemente como los otros métodos de desplazamiento y reducción.
- La clase de gramáticas que pueden analizarse con los métodos LR es un supraconjunto de la clase de gramáticas que se pueden analizar con analizadores sintácticos predictivos.
- Un analizador sintáctico LR puede detectar un error sintáctico tan pronto como sea posible hacerlo en un examen de izquierda a derecha de la entrada.

El principal inconveniente del método es que supone demasiado trabajo construir un analizador sintáctico LR a mano para una gramática de un lenguaje de programación típico. Se necesita una herramienta especializada un generador de analizadores sintácticos LR. Por fortuna, existen disponibles estos generadores. El programa YACC. Con este generador se puede escribir una gramática independiente del contexto y el generador produce automáticamente un analizador sintáctico para dicha gramática. Si la gramática contiene ambigüedades u otras construcciones difíciles de analizar en un examen de izquierda a derecha de la entrada, el generador puede localizar dichas construcciones e informar al diseñador del compilador de su presencia.

Después de estudiar la operación de un analizador sintáctico LR para una gramática. El primer método, llamado LR sencillo (SLR,en ingles) es el más fácil de implantar, pero el menos poderoso de los tres. Puede que no consiga producir una tabla de análisis sintáctico para algunas gramáticas que otros métodos si consiguen. El segundo método, llamado LR canónico, es el más poderoso y costoso. El tercer método, llamado LR con examen por anticipado (LALR, en ingles), esta entre los otros dos en cuanto a poder y costo. El método LALR funciona con las gramáticas de la mayoría de los lenguajes de programación y, con un poco de esfuerzo, se puede implantar en forma eficiente. En esta sección se consideran más adelante algunas técnicas para comprimir el tamaño de una tabla de análisis sintáctico LR.

El algoritmo de análisis sintácticos LR.

En la siguiente figura se muestra la forma esquemática de un analizador sintáctico LR. Consta de una entrada, una salida, una pila, un programa conductor y una tabla de análisis sintáctico con dos partes (acción e ir_a). El programa conductor es el mismo para todos los analizadores sintácticos LR; solo cambian las tablas de un analizador a otro. El programa analizador lee caracteres de un buffer de entrada de uno en uno. El programa utiliza una pila para almacenar una cadena de la forma $S_0X_{1S1}X_{2S2}\ldots X_{mSm}$, donde S_m esta en la cima. Cada X es un símbolo gramatical y cada S_i es un símbolo llamado *estado*. Cada símbolo de estado resume la información contenida

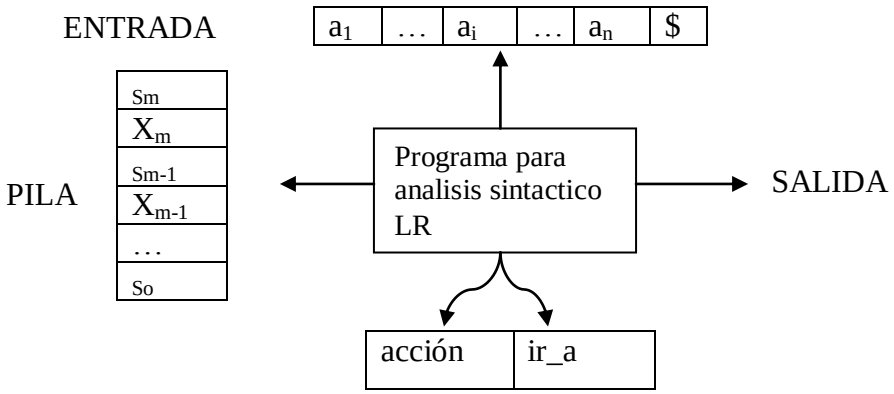


Fig. 5.11. Modelo de analizador sintáctico LR.

debajo de el en la pila, y se usan la combinación del símbolo de estado en la cima de la pila y el símbolo en curso de la entrada para indexar la tabla de análisis sintáctico y determinar la decisión de desplazamiento a reducción del analizador. En una implantación no es necesario que los símbolos de la gramática aparezcan en la pila; sin embargo, siempre se incluirán en las siguientes exposiciones, para facilitar la explicación del comportamiento de un analizador sintáctico LR.

La tabla de análisis sintáctico consta de dos partes, la función acción, que indica una acción del analizador, y la función ir_a, que indica las transiciones entre estados. El programa que maneja el analizador sintáctico LR se comporta como sigue: determina S_m, el estado de la cima de la pila, y ai, el símbolo en curso de la entrada. Después consulta la entrada *acción*$[S_m,a_i]$ de la tabla de acciones del analizador para el estado S_m, y la entrada a_i, que puede tener uno de estos cuatro valores:

1. Desplazar S, donde S es un estado,
2. Reducir por una producción gramatical $A \rightarrow \beta$,
3. Aceptar y
4. Error.

La función ir_a toma un estado y un símbolo gramatical como argumentos y produce un estado. Se verá que la función ir_a de una tabla de análisis sintáctico construida a partir de una gramática G utilizando el método SLR, LR canónico o LALR es la función de transiciones de un autómata finito determinista que reconoce los prefijos viables de G.

Recuérdese que los prefijos viables de G son aquellos prefijos de forma de frase derecha que pueden aparecer en la pila de un analizador sintáctico por desplazamiento y reducción, porque no sobrepasan el mango situado más a la derecha. El estado inicial de esta AFD es el estado puesto inicialmente en la cima de la pila del analizador LR.

Una *configuración* de un analizador sintáctico LR es un par cuyo primer componente es el contenido de la pila, y el segundo, la entrada todavía sin procesar
($S_0X_1S_1X_2S_2\ldots X_mS_m,a_ia_i{+}1\ldots a_n\$$).

Esta configuración representa la forma de frase derecha
$X_1X_2\ldots X_ma_ia_i{+}1\ldots a_n$

Fundamentalmente de la manera en que lo haría un analizador sintáctico por desplazamiento y reducción; solo es nueva la presencia de los estados en la pila. El siguiente movimiento del analizador se determina leyendo ai, el símbolo de la entrada en curso, y Sm, el estado del tope de la pila, y consultando después la entrada acción $[S_m,a_i]$ de la tabla de acciones del analizador. Las configuraciones obtenidas después de cada uno de los cuatro tipos de movimiento son las siguientes:

1. Si acción $[_{Sm},a_i]$= desplazar $_S$, el analizador ejecuta un movimiento de desplazamiento, entrando en la configuración
   ($_{SO}X_{1S1}X_{2S2}\ldots X_{mSm}a_{iS},a_i{+}1\ldots a_n\$$)

   Aquí, el analizador ha desplazado a la pila al símbolo de entrada en curso a y al siguiente estado S, que esta dado en acción$[_{Sm},a_i]$; a_{i-1} se convierte en el símbolo de entrada en curso.

2. Si acción [Sm,ai]= reducir A→β, entonces el analizador ejecuta un movimiento de reducción, entrando , en la configuración
   ($_{So}X_{1S1}X_2S_2\ldots X_{mr\text{-}Sm\text{-}r}A\ _S,a_ia_{i+1}\ldots a_n\$$)

donde s=ir_a[S_m-r,A], y r es la longitud de β, el lado derecho de la producción. Aquí el analizador extrajo primero 2r símbolos de la pila (r símbolos de estados y r símbolos de la gramática), exponiendo el estado Sm-r . Luego introdujo A, el lado izquierdo de la producción, y s, la entrada de ir-a[S_m-r,A], en la pila. En un movimiento de reducción no se modifica el símbolo de entrada en curso. Para los analizadores LR que se construirán, X_m-r+1…Xm, la secuencia de símbolos gramaticales extraídos de la pila, siempre concordaran con β, el lado derecho de la producción con que se efectúa la reducción.

Después de un movimiento de reducción, se genera la salida de un analizador después de un movimiento de reducción se genera la salida de un analizador, LR ejecutando la

acción semántica asociada a la producción con que se efectúan la reducción. Por el momento, se asumirá que la salida consiste únicamente en imprimir la producción con que se efectúa la reducción.

3. Si acción[S_m,a_i]= aceptar, el análisis sintáctico ha terminado.
4. Si acción [S_m,a_i]=error, el analizador ha descubierto un error y llama a una rutina de recuperación de errores.

El algoritmo de análisis sintáctico LR se resume más adelante. Todos los analizadores sintácticos LR se comportan de esa forma; la única diferencia entre uno y otro es la información de los campos de acción y de transición de la tabla de análisis sintáctico.

Algoritmo 5.2. Análisis sintáctico LR.

Entrada. Una cadena de entrada W y una tabla de análisis sintáctico LR con las funciones acción e ir_a para la gramaticaG.

Salida. Si W esta en L(G), un análisis sintáctico ascendente de W; de lo contrario se indica error.

Método. Inicialmente, S_0 esta en la pila del analizador sintáctico, donde S_0 es el estado inicial, y W$ esta en el buffer de entrada. El analizador ejecuta entonces el programa de la figura siguiente hasta encontrar una acción de aceptación de error.

Ejemplo En la figura 5.12. se muestran las funciones acción e *ir_a* del análisis sintáctico de una tabla de análisis sintáctico LR para la siguiente gramática para expresiones aritméticas con los operadores binarios + y *:

Gramática G10:

(1) E →E +T
(2) E→T
(3) T→T*F
(4) T→F
(5) F→(E)
(6) F→**id**

Los códigos de las acciones son:

1. d*i* significa desplazar y meter en la pila el estado *i*,
2. r*j* significa reducir por la producción con numero *j*,
3. Acep significa aceptar,
4. El espacio en blanco significa error.

Programa para análisis sintáctico LR.

apuntar *ae* al primer símbolo de *w*$;
repeat forever begin
sea *s* el estado en la cima de la pila y
a el símbolo apuntado por *ae*;
if acción [*s, a*]= desplazar s' **then begin**
meter a y después *s'* en la cima de la pila;
avanzar *ae* al siguiente símbolo de entrada
end
else if *acción* [*s,a*] = reducir *A*-->β **then begin**
sacar 2 * | β | símbolos de la pila;
sea *s'* el estado que ahora esta en la cima de la pila;
meter A y después *ir_a* [*s', A*] en la cima de la pila;
emitir la producción a-->β
end
else if *accion* [*s,a*] = aceptar **then**
return
else *error*()
end

	acción						***ir_a***		
ESTADO	**Id**	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
0	d5			d4			1	2	3
1		d6				acep			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Fig. 5.12. Tabla de análisis sintáctico para la gramática de expresiones.

Obsérvese que el valor de *ir_a*[*s,a*] para el terminal *a* se encuentra en el campo acción conectado con la acción de desplazar en la entrada a para el estado *s*. El campo de *ir_a* da *ir_a*[*s,A*] para los no terminales *A*. Asimismo, téngase en cuenta que aun no se ha explicado cómo se seleccionaron las entradas de la figura 5.12; este aspecto se considerará más adelante.

PILA	ENTRADA	ACCION
(1) 0	**id** * **id** + **id** $	desplazar
(2) 0 **id** 5	* **id** + **id** $	reducir por F → id
(3) 0 F 3	* **id** + **id** $	reducir por T→F
(4) 0 T 2	* **id** + **id** $	desplazar
(5) 0 T 2 * 7	**id** + **id** $	desplazar
(6) 0 T 2 * 7 **id** 5	+ **id** $	reducir por F→id
(7) 0 T 2 * 7 F 10	+ **id** $	reducir por T→T*F
(8) 0 T 2	+ **id** $	reducir por E→T
(9) 0 E 1	+ **id** $	desplazar
(10) 0 E 1 +6	**id** $	desplazar
(11) 0 E 1 + 6 **id** 5	$	reducir por F→id
(12) 0 E 1 + 6 F 3	$	reducir por T→F
(13) 0 E 1 + 6 T 9	$	E→E + T
(14) 0 E 1	**$**	aceptar

Fig. 5.13. Movimientos del analizador sintáctico LR con la entrada **id** * **id** + **id**.

Con la entrada **id** * **id** + **id**, en la figura 5.13 se muestra la secuencia de contenidos de la pila y de la entrada. Por ejemplo, en la línea (1) el analizador LR está en el estado 0, siendo **id** el primer símbolo de entrada. La acción en la fila 0 y columna id del campo acción de la tabla anterior a esta es d5, que significa desplazar y tapar la cima de la pila con el estado 5. Esto es lo que ha ocurrido en la línea (2): el primer componente léxico id y el símbolo del estado 5 han sido introducidos en la pila y se ha eliminado id de la entrada.

Entonces, * se convierte en el símbolo de entrada en curso y la acción del estado 5 con la entrada * es reducir por F→ **id**. Se extraen dos símbolos de la pila (un símbolo de estado y un símbolo gramatical). El estado 0 queda expuesto en la cima de la pila. Como el ir del estado 0 en F es 3, se introducen F y 3 en la pila. Ya se tiene la configuración de la línea (3). Los movimientos restantes se determinan de manera similar.

Gramáticas LR.

¿Cómo se construye una tabla de análisis sintáctico LR para una determinada gramática? Una gramática para la que se puede construir una tabla de análisis sintáctico se denomina *gramática* LR. Hay gramáticas independientes del contexto que no son LR, pero en general se pueden evitar en las construcciones típicas de los lenguajes de programación. Intuitivamente, para que una gramática sea LR basta con que un analizador sintáctico por desplazamiento y reducción que opere de izquierda a derecha pueda reconocer los mangos cuando aparezcan en la cima de la pila.

Un analizador LR no tiene que examinar la pila completa para saber cuando aparecen los mangos en la cima. Por el contrario, el símbolo del estado en la cima de la pila contiene toda la información necesaria. Es un hecho curioso que, si se puede reconocer un mango conociendo solo los símbolos gramaticales de la pila, entonces existe un autómata finito que puede, leyendo los símbolos gramaticales de la pila de arriba abajo, determinar el mango, si existe, que esta en el tope de la pila. La función ir_a de una tabla de análisis sintáctico LR es esencialmente dicho autómata finito. Sin embargo, el autómata no necesita leer la pila para cada movimiento. El símbolo estado almacenado en la cima de la pila es el estado en que estaría el autómata finito reconocedor de los mangos si hubiera leído los símbolos gramaticales de la pila desde abajo hasta la cima. Por tanto, el analizador sintáctico LR puede determinar a partir del estado de la cima de la pila todo lo que se necesita saber sobre lo que hay en ella.

Otra fuente de información que puede utilizar un analizador LR como ayuda para tomar las decisiones de desplazamiento y reducción son los k símbolos siguientes de entrada. Los casos en que k=0 o k=1 tienen interés prácticos, y aquí solo se consideraran los analizadores sintácticos LR con $k \leq 1$. Por ejemplo, la tabla de acciones de la tabla de análisis anterior utiliza un símbolo de examen por anticipado. Una gramática que se puede analizar mediante un analizador sintáctico LR que examina hasta k símbolo de entrada en cada movimiento se denomina gramática LR(k).

Existe una diferencia significativa entre las gramáticas LL y las LR. Para que una gramática sea LR(k), hay que ser capaz de reconocer la presencia del lado derecho de una producción, habiendo visto todo lo que deriva de dicho lado derecho con k símbolos de examen por anticipado. Este requisito es mucho menos riguroso que el de las gramáticas LL(k), donde hay que ser capaz de reconocer el uso de una producción viendo solo los primeros k símbolo de los que se deriva su lado derecho. Por consiguiente las gramáticas LR pueden describir más lenguajes que las gramáticas LL.

Construcción de tablas de Análisis Sintácticos SLR.

A continuación se muestran cómo construir una tabla de análisis sintáctico SLR a partir de una gramática. se darán tres métodos, que tienen distintos grados de poder y facilidad de aplicación. El primero, llamado “LR sencillo” (o SLR, en ingles), es el más débil de los tres en cuanto al número de gramática para las que funcionan con éxitos, pero es el más fácil de implantar. La tabla de análisis sintáctico construidas con este método se denominara tabla SLR y un analizador LR que utilice una tabla de análisis SLR se denominara analizador sintáctico SLR se denomina gramática SLR. Los otros dos

métodos amplían el método SLR con información de examen por anticipado, así que el método SLR es un buen punto de partida para estudiar el análisis sintáctico LR.

Un *elemento del análisis sintáctico LR(0) (elemento,* para abreviar*)* de una gramática G es una producción de G con un punto en alguna posición del lado derecho. Por tanto, la producción A→ XYZ produce los cuatro elementos.

A→.XYZ
A→X.YZ
A→XY.Z
A→XYZ.

La producción A→ϵ genera solo un elemento, A→ . Un elemento se puede representar mediante dos enteros, el primero de los cuales da el número de la producción, y el segundo, la posición del punto. Intuitivamente, un elemento indica hasta donde se ha visto una producción en un momento dado del proceso del análisis sintáctico. Por ejemplo, el primer elemento de arriba indica que se espera ver a continuación en la entrada una cadena derivable de XYZ. El segundo elemento indica que se acaba de ver en la entrada una cadena derivable de X y que a continuación se espera ver una cadena derivable de YZ.

La idea central del método SLR es construir primero a partir de la gramática un autómata finito determinista para reconocer los prefijos viables. Los elementos se agrupan en conjuntos, que da lugar a los estados del analizador sintáctico SLR. Los elementos se pueden considerar como los estados de un AFN que reconoce los prefijos viables, y el "agrupamiento" es en realidad la construcción de subconjunto estudiadas anteriormente.

Una serie de conjunto LR (0), que se denomina colección *canoníca* LR (0), proporciona la base para construir analizadores sintácticos SLR. Para construir la colección canónica LR (0) para una gramática se define una gramática aumentada y dos funciones, *cerradura* e ira.

Si G es una gramática con símbolo inicial S, entonces G', la *gramática aumentada* para G, es G con un nuevo símbolo inicial S'y la producción S' → S. el propósito de esta nueva producción inicial es indicar al analizador cuando debe detener el análisis sintáctico y anunciar la aceptación de la cadena. Es decir, la aceptación se produce cuando, y solo cuando, el analizador esta apunto de reducir por S'→ S.

La operación cerradura

Si I es un conjunto de elementos para una gramática G, entonces *cerradura* (I) es el conjunto de elemento construido a partir de I por las dos reglas:

1. Inicialmente, todo elemento de I se añade a *cerradura* (I).
2. Si A→α. B β esta en *cerradura* (I) y B→γ es una producción, entonces añádase el elemento B→. γ a *cerradura*(I), si todavía no esta ahí. Se aplica esta regla hasta que no se puedan añadir más elementos a *cerradura*(I).

Intuitivamente, si A→α. B β esta en *cerradura* (I) indica que, en algún momento del proceso de análisis sintáctico, se cree posible ver a continuación una cadena derivable de B como entrada. Si B→γ es una producción, también se espera ver una subcadena derivable de γ en este punto. Por esta razón se incluye B→. γ en *cerradura*(I).

Considérese la gramática G11 de expresiones aumentadas:
E'→E
E→E+T|T
T→T*F|F
F→(E)|id

Si I es el conjunto de un elemento {[E'→.E]}, entonces *cerradura* (I) contienen los elementos
E'→.E
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.**id**

Aquí, E'→.E se coloca en *cerradura*(I) por la regla 1. Como hay una E inmediatamente a la derecha de un punto, por la regla 2 se añaden las producciones de E con puntos en el extremos izquierdo, es decir, E→.E + T y E→. T. Ahora hay una T inmediatamente a la derecha de un punto, así que se añade T→ .T*F y T→.F. A continuación la F a la derecha de un punto obliga a añadir F→.(E) y F→id. Por la regla 2 no se coloca más elementos dentro de *cerradura* (I).

Se puede calcular la función *cerradura* como se muestra en el segmento de código del Cálculo de *cerradura.* Una forma apropiada de implanta la función *cerradura* es mantener una matriz booleana *añadida*, indexada por los no terminales de G, de forma que a *añadida* [B] se le asigna **true** siempre y cuando se añadan los elementos B→ .γ para cada producción de B B→ γ.

Calculo de cerradura.

```
function cerradura(I)
begin
        J:=I;
        repeat
                for cada elemento A→α.Bβ en J y cada producción
                   B→γ de G tal que B→.γ no este en J do
                        añadir B→. γ a J
         until no se puede añadir más elementos a J;
        return J
end
```

LABORATORIO DE DISEÑO DE COMPILADORES

INTRODUCCIÓN

Conforme se va abordando la teoría es importante que el alumno vaya realizando ciertos ejercicios para llevar esa teoría a la práctica que le permita afianzar y reforzar con una visión más clara y amplia los conocimientos que se van adquiriendo.

Para el laboratorio se cuentan con cuatro horas semanales, en de las cuales, antes de iniciar una nueva práctica, el profesor deberá exponerla así como la teoría adicional que se necesite para resolverla, también durante estas cuatro horas el profesor debe estar presente para resolver cualquier duda o problema que se les presente a los estudiantes.

Las prácticas se realizarán en grupos de tres estudiantes como máximo.

La evaluación del laboratorio corresponde al 40% de las notas parciales de la asignatura. Los estudiantes deberán realizar todas las prácticas establecidas en tiempo y forma para poder tener derecho al porcentaje del laboratorio. El estudiante deberá mostrar cada práctica funcionando, y entregar un documento impreso que contendrá la explicación de la solución, el código comentado y conclusiones. Todas las prácticas impresas tendrán un valor del 10% complementando con un 30% en una defensa grupal el 40% correspondiente a las prácticas de laboratorios.

PLANIFICACIÓN TEMPORAL

Como ya se ha mencionado con anterioridad, para el desarrollo de las prácticas de laboratorio se cuenta con dos sesiones de dos horas por semana, para un total de 56 horas a lo largo de las 14 semanas netas con las que se cuenta para el laboratorio. El número de horas asignadas para cada práctica ha sido obtenido en base a la complejidad relativa y al tiempo total disponible.

DESARROLLO DE LAS PRÁCTICAS

PRÁCTICA NO. 1

Implementación de una estructura de datos para simular una Tabla de Símbolos y Atributos.

Objetivo:

- Reutilizar una librería que implementa una estructura de datos llamada pila de lista.

Desarrollo:

1. Se les proporcionará un archivo .h donde se encuentra el código que implementa las funciones básicas relacionadas con pilas y listas.
2. El estudiante elaborará un archivo .c que realice la simulación de una TSA de un compilador.

a) Comprobar que la estructura está vacía.
b) Hacer las llamadas para llenar la estructura.
c) Mostrar el contenido de la estructura en diferentes momentos del llenado.
d) Hacer las llamadas para ir vaciando la estructura.
e) Mostrar el contenido de la estructura en diferentes momentos del vaciado.
f) Asegurarse que la estructura quede completamente vacía al finalizar el programa.

PRÁCTICA NO. 2

Objetivo:

- Escribir un programa utilizando código ANSI C para implementar un Scanner.

Desarrollo:

Nuestra primera visión del programa llamado scanner es la siguiente.
El scanner lee carácter por carácter los símbolos de entrada en el programa fuente.
El scanner reconoce o agrupa palabras con significado para el lenguaje, elimina la información innecesaria como espacio en blanco, comentarios, etc. Al finalizar está práctica el estudiante mostrará la salida del scanner para el siguiente programa:

```
begin
var int numero, suma;
procedure  leer
begin
        while (numero <>0)do
       suma:=suma+numero;
       read(numero);
od;
end; #
suma:=0;
read(numero);
call leer;
write(suma);
end.
```

La salida del scanner debe mostrar

1. Una lista el número de tokens que componen el programa.
2. Informar cuantas lineales tiene el programa.

PRÁCTICA NO. 3

Elaborar una aplicación que simule las funciones que realiza el Scanner como son:

1- Identificar códigos inválidos para el lenguaje definido, emitir mensaje de error y la línea donde se encuentra el código desconocido.
2- Listar los tokens contenidos en el input de entrada mostrar además el número de línea donde se encuentra ubicado el token.

Programa fuente en el cual se realizará el análisis léxico

```
begin#
var int numero, suma;
proc leer
begin
      while(numero<>0)do
            suma:=suma+numero;
?
read(suma);
suma:0;#
read(numero);
call leer;
write(suma);
end.
```

PRÁCTICA NO. 4

DISEÑO DE UN COMPILADOR CON C++
EL PARSER

Objetivos:

- Diseñar un parser para un compilador
- Diseñar un compilador para un lenguaje funcional
- Escribir el código en lenguaje C++
- Implementar la teoría estudiada en clase en nuestro proyecto
- Exponer y Defender por grupo el proyecto

Desarrollo:

Un PARSER debe tener una manera para responder a los errores de sintaxis encontrados en el medio del parseo. El parseo para SLANG AMPLIADO usa la técnica recursiva descendente, requiriendo que cada no Terminal en la gramática tenga un procedimiento separado escrito para este. Usar esta técnica de parseo potencialmente requiere que el manejo de errores sea integrado con cada uno de los procedimientos de parseo. De examinar nuestros procedimientos de parseo para SLANG AMPLIADO, es evidente que los errores de sintaxis puede ser detectado en cualquier de dos maneras: **match()** puede fallar al tratar de identificar el token correcto del programa fuente, o un procedimiento de parseo que examina **next_token()** en un **swith()** or **if** puede fallar de hallar un token aceptable. En el ultimo caso, las rutinas de parseo de SLANG AMPLIADO llama a la rutina **syntax_error()** y **match()** y **syntax_error()** deben ser designada para tomar algunas acciones que van a permitir al parser continuar.
match() puede ser diseñado como una función booleana en orden de que indique cuando el token requerido fue encontrado en el input, pero esta alternativa puede grandemente complicar cada procedimiento de parseo con llamadas a este. Una alternativa simple podría ser que **match()** maneje errores, simplemente pretendiendo que este vio el token correcto. En tal caso, debemos decidir si **match()** debe consumir el token incorrecto como si este encontró el token deseado, a como lo hace en el caso de una comprobación correcta.
Si el segundo tipo de errores de sintaxis es encontrado, no hay un simple mecanismo posible, cualquier elemento del conjunto de tokens es aceptable para continuar el parseo. Manejar estos errores requerirá reprogramación explicita de algunas de las rutinas de parseo. Esto sugiere de manera general que procedimientos como **statement()** pueden ser cambiado para manejar errores de sintaxis.

Algunas funciones del parser en la librería parser.h

```
void program_declaration(void);
void block(void);
void declaration_part(void);
void constant_declaration(void);
void type_declarer(void);
```

```
  void variable_declaration(void);
  void PROCEDURE_declaration(void);
  void statement(void);
```

Completarlas

```
void Sintaxis_Error(tokens t, int l)
 {
    printf("Error de sintaxis:%s Linea:%d\n",tokenName[t],l");
      return;
}
```

(CFG para el lenguaje SLANG A.)

program_declaration → block period_token

begin_token→{declaration_part|variable_declaration}*

(CFG para el lenguaje SLANG A.)

void program_declaration(void)

```
{
token_actual=cabLT;
block();
match(PERIOD_);
}
```

void block()

```
{
        match(BEGIN_)
          declaration_part();
          statement_part();
          match(END_);
        }
```

Input de entrada

```
begin
var a,b,c
a:=2;
b:=3;
c=a+b;
end.
```

FIN DEL ESCANEO
Desea visualizar la lista de tokens?(S/N)
Presione una tecla para continuar…
Numero de línea: 1
El nombre del tokens: BEGIN_
Presione una tecla para continuar…
Numero de línea: 2
El nombre del tokens: VAR_
El nombre del tokens: ID_ El nombre del identificador: a
El nombre del tokens: LIST_
El nombre del tokens: ID_ El nombre del identificador: b
El nombre del tokens: LIST_
El nombre del tokens: ID_ El nombre del identificador: c
Presione una tecla para continuar…
Numero de línea: 3
El nombre del tokens: ID_ El nombre del identificador: a
El nombre del tokens: ASSING_
El nombre del tokens: NUMBER_ Su valor es ==> 2
El nombre del tokens: SEPARATOR_
Presione una tecla para continuar…
Numero de línea: 4
El nombre del tokens: ID_ El nombre del identificador: b
El nombre del tokens: ASSIGN_
El nombre del tokens: NUMBER_ Su valor es ==> 3
Presione una tecla para continuar…
Numero de línea: 5
El nombre del tokens: ID_ El nombre del identificador: b
El nombre del tokens: EQUAL_
El nombre del tokens: ID_ El nombre del identificador: b
El nombre del tokens: PLUSOP_
El nombre del tokens: ID_ El nombre del identificador: b
El nombre del tokens: SEPARATOR_
Presione una tecla para continuar…
Numero de línea: 6
El nombre del tokens: END_
El nombre del tokens: PERIOD_

Errores encontrados:
Error de sintaxis: ID_ Linea: 2, token esperado SEPARATOR_, o LIST_
Error de sintaxis: EQUAL_ Linea: 5

FIN DEL PARSEO
Press any key to continue

PRÁCTICA NO. 5

Elaborar una aplicación para calcular el Conjunto Primero de la gramática del SLANG AMPLIADO.

Dicha aplicación pedirá como entrada el nombre del archivo ó programa fuente en donde se calculará no solo el Conjunto Primero sino también una lista de Tokens y símbolos desconocidos, un resumen general de un input de entrada, y errores de parseo.

BIBLIOGRAFÍA

- Compilers Principles, Techniques, & Tolos – Second Edition.
  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman.

- Compiladores Principios, técnicas y herramientas.
  Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman.

- Pascal introducción
  Jim Welsh - The University of Manchester Institute of Science and Technology, England.
  John Elder - Quee ‘s University of Belfast, Northern Ireland.
  ISBN 968–880–028–7

- Construcción de compiladores, Principios y práctica
  Kenneth C. Louden

ANEXOS

SOLUCIÓN DE LA PRÁCTICA NO. 1

```
#define VERDADERO 1
#define FALSO 0

//Inicializa el nivel actual
int Nivel =0;

//Estructura del Record semántico
typedef struct RecordSemantico
{
	char name[10];
	int value;
	//la estructura del record semántico se modifica
	//de acuerdo a lo que se exponga en clase teóricas
}Record_Semantico;

//Estructura de los nodos de las listas
typedef struct nodos_lista nodo_lista;
struct nodos_lista
{
	Record_Semantico rs;
	nodo_lista * siguiente;
};

//Estructura de los nodos de la Pila
typedef struct nodos_pila nodo_pila;
struct nodos_pila
{
	int nivel;
	nodo_lista *lista;
	nodo_pila *siguiente;
};

//Definimos la Tabla de Simbolos y Atributos
nodo_pila *TSA = NULL;

//FUNCIONES PARA LA LISTA

//Funciones para crear un nuevo bloque de datos en memoria para la lista
nodo_lista *Nuevo_elemento_lista(void)
{
	nodo_lista *a=(nodo_lista *)malloc(sizeof(nodo_lista));
	if(a==NULL)
	{
		printf("ERROR: No hay memoria!");
```

```
                exit(0);
        }
        return(a);
}
//Funcion para buscar un elemento en la lista
int Buscar_elemento_lista(nodo_lista *lista, char *cadenabuscar)
{
        while(lista!=NULL)
        {
                if(strcmp(lista->rs.name,cadenabuscar)==0)
                        return(VERDADERO);
                lista=lista->siguiente;
        }
        return(FALSO);
}

void Liberar_memoria(nodo_pila **);

//Función para enlazar un nuevo elemento en la lista
void Agregar_elemento_lista(nodo_lista **lista, Record_Semantico a)
{
        nodo_lista * nuevo, *actual;
        nuevo=Nuevo_elemento_lista();
        nuevo->rs=a;
        nuevo->siguiente=NULL;
        actual=*lista;
        if(Buscar_elemento_lista(actual, a.name))
        {
                free(nuevo);
                printf("ERROR: Nombre de ID duplicado");
                Liberar_memoria(&TSA);
                exit(0);
        }
        if(actual==NULL)
        {
                *lista=nuevo;
                return;
        }

        //Mueve el puntero hasta el final
        while((actual->siguiente)!=NULL)
                actual=actual->siguiente;
        actual->siguiente=nuevo;
}

//Función que imprime la lista completa
void Imprimir_lista(nodo_lista *lista)
{
        if(lista==NULL)
        {
```

```
		printf("La lista esta vacia\n");
		return;
	}
	while(lista!=NULL)
	{
		printf("%s %d\n",lista->rs.name,lista->rs.value);
		lista=lista->siguiente;
	}
}

//Función que borra la lista de la memoria
void Borrar_lista(nodo_lista **lista)
{
	nodo_lista *temp;
	temp=*lista;
	while(temp!=NULL)
	{
		*lista=(*lista)->siguiente;
		free(temp);
		temp=*lista;
	}
}

//FUNCIONA PARA LA PILA
//Función para crear un nuevo nodo en la pila
nodo_pila *Nuevo_elemento_pila(void)
{
	nodo_pila *a=(nodo_pila*)malloc(sizeof(nodo_pila));
	if(a==NULL)
	{
		printf("ERROR: No hay memoria!");
		exit(0);
	}
	return(a);
}

//Función para verificar si la Pila está vacía

int Esta_vacia(nodo_pila * top)
{
	if(top==NULL)
		return(VERDADERO);
	else
		return(FALSO);
}

//Función PUSH: Insertar un bloque de memoria y lo enlaza, pero no agrega lista
void Push(nodo_pila **top)
{
	nodo_pila *nuevo;
```

```
	nuevo=Nuevo_elemento_pila();
	Nivel++;
	nuevo->nivel=Nivel;
	nuevo->lista=NULL;
	nuevo->siguiente=*top;
	*top=nuevo;
}

//Función POP: borra la lista del tope de la lista y desaparece ese bloque de la pila.
void Pop(nodo_pila ** top)
{
	nodo_pila * temp;
	if(Esta_vacia(*top))
	{
		printf("ERROR: La pila esta vacia\n");
		return;
	}
	temp=*top;
	*top=(*top)->siguiente;
	Borrar_lista(&temp->lista);
	free(temp);
	Nivel--;
}
//Función para imprimir la pila completa
	void Recorrer_pila(nodo_pila *top)
	{
		int tempNivel=Nivel;
		if(Esta_vacia(top))
		{
			printf("ERROR: La pila esta vacia\n");
			return;
		}
		while(top!=NULL)
		{
			printf("NIVEL %d\n",tempNivel--);
			printf("pila->nivel %d\n",top->nivel);
			Imprimir_lista(top->lista);
			top=top->siguiente;
		}
	}

	//Funcion para liberar toda la memoria asignada dinámicamente
	void Liberar_memoria(nodo_pila **top)
	{
		while((*top)!=NULL)
			Pop(top);
	}


#include<stdio.h>
```

```
#include<stdlib.h>
#include<string.h>
#include "tsa.h"


main()
{
	//Declaramos un Record_Semantico temporal
	Record_Semantico temp;

	system("cls");
	printf("\n\n\nAun no se han insertado elementos\n");
	Recorrer_pila(TSA);
	system("pause");

	Push(&TSA);

	strcpy(temp.name,"prueba1");
	temp.value=1;
	Agregar_elemento_lista(&TSA->lista,temp);

	strcpy(temp.name,"prueba2");
	temp.value=2;
	Agregar_elemento_lista(&TSA->lista,temp);

	printf("\n\n\nSe insertaron mas elementos y se lista de nuevo\n");
	Recorrer_pila(TSA);
	system("pause");


	strcpy(temp.name,"prueba3");
	temp.value=3;
	Agregar_elemento_lista(&TSA->lista,temp);

	strcpy(temp.name,"prueba4");
	temp.value=4;
	Agregar_elemento_lista(&TSA->lista,temp);

	printf("\n\n\nSe insertaron mas elementos y se lista de nuevo\n");
	Recorrer_pila(TSA);
	system("pause");

	Push(&TSA);

	strcpy(temp.name,"prueba5");
	temp.value=5;
	Agregar_elemento_lista(&TSA->lista,temp);

	strcpy(temp.name,"prueba6");
	temp.value=6;
```

```
Agregar_elemento_lista(&TSA->lista,temp);

strcpy(temp.name,"prueba7");
temp.value=7;
Agregar_elemento_lista(&TSA->lista,temp);

printf("\n\n\nSe insertaron mas elementos y se lista de nuevo\n");
Recorrer_pila(TSA);
system("pause");

strcpy(temp.name,"prueba8");
temp.value=8;
Agregar_elemento_lista(&TSA->lista,temp);

strcpy(temp.name,"prueba9");
temp.value=9;
Agregar_elemento_lista(&TSA->lista,temp);

strcpy(temp.name,"prueba10");
temp.value=10;
Agregar_elemento_lista(&TSA->lista,temp);

printf("\n\n\nSe insertaron mas elementos y se lista de nuevo\n");
Recorrer_pila(TSA);
system("pause");

Push(&TSA);

strcpy(temp.name,"prueba11");
temp.value=11;
Agregar_elemento_lista(&TSA->lista,temp);

strcpy(temp.name,"prueba12");
temp.value=12;
Agregar_elemento_lista(&TSA->lista,temp);

strcpy(temp.name,"prueba13");
temp.value=13;
Agregar_elemento_lista(&TSA->lista,temp);

Recorrer_pila(TSA);
system("pause");

printf("\n\n\nSe hará un pop\n");
Pop(&TSA);

Recorrer_pila(TSA);
system("pause");

strcpy(temp.name,"prueba14");
```

```
        temp.value=14;
        Agregar_elemento_lista(&TSA->lista,temp);

        strcpy(temp.name,"prueba15");
        temp.value=15;
        Agregar_elemento_lista(&TSA->lista,temp);

        printf("\n\n\nSe insertaron mas elementos y se lista de nuevo\n");

        Recorrer_pila(TSA);
        system("pause");

        printf("\n\n\nSe hará un po\n");
        Pop(&TSA);

        Recorrer_pila(TSA);
        system("pause");

        //Queda dos niveles y se leberará todo la memoria asignada
        Liberar_memoria(&TSA);

}
```

SOLUCIÓN DE LA PRÁCTICA NO. 2

```
/*

#include<ctype.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void main(void)
{
	FILE *ptrToFile;
	char NombreArchivo[12];
	char caracteractual;
	int cuentatotal=0;
	int cuentaespacios=0;
	int cuentalineas=0;

	printf("Nombre del archivo:");
	gets(NombreArchivo);
	if((ptrToFile=fopen(NombreArchivo,"r"))==NULL)
	{
		printf("El archivo no se puede abrir\n");
		exit(0);
	}

	printf("El archivo se puede abrir\n");
	while(!ferror(ptrToFile) && !feof(ptrToFile))
	{caracteractual=fgetc(ptrToFile);
	if(isspace(caracteractual))
		cuentaespacios++;
	if(caracteractual=='\n')
		cuentalineas++;
	cuentatotal++;

	}

	printf("Cuenta Total =%d\n",cuentatotal);
	printf("Numero de espacios =%d\n",cuentaespacios);
	printf("Numero de lineas = %d\n",cuentalineas);
	fclose(ptrToFile);
	system("pause");
}
*/

#include<ctype.h>
#include<stdio.h>
#include<stdlib.h>
```

```
#include<string.h>

void main(void)
{
	FILE *ptrToFile;
	char NombreArchivo[12];
	char caracteractual, temp;
	int cuentaespacios=0;
	int cuentalineas=0;
	int cuentatokens=0;

	printf("Nombre del archivo:");
	gets(NombreArchivo);
	if((ptrToFile=fopen(NombreArchivo,"r"))==NULL)
	{
		printf("El archivo no se puede abrir\n");
		exit(0);
	}

	printf("El archivo se puede abrir\n");

	//CICLO PRINCIPAL DEL SCANNER

	while(!ferror(ptrToFile) && !feof(ptrToFile))
	{
		//Leemos el caracter actual
		caracteractual=fgetc(ptrToFile);
		//Reconocemos si es un fin de linea
		if(caracteractual=='\n')
		{
			cuentalineas++;
			continue;
		}
		//Reconocemos si es un espacio en blanco
		if(isspace(caracteractual))
		{
			cuentaespacios++;
			continue;
		}
		//Reconocemos si es un identificador
		if(isalpha(caracteractual))
		{
			temp=fgetc(ptrToFile);
			while(isalnum(temp))
			{
				temp=fgetc(ptrToFile);
			}
			ungetc(temp,ptrToFile);
			cuentatokens++;
			continue;
```

```
}

//Reconocemos si es un número entero
if(isdigit(caracteractual))
{
	temp=fgetc(ptrToFile);
	while(isdigit(temp))
	{
		temp=fgetc(ptrToFile);
	}
	ungetc(temp,ptrToFile);
	cuentatokens++;
	continue;
}
//Reconocemos si es un parentesis de apertura
if(caracteractual=='(')
{
	cuentatokens++;
	continue;
}
//Reconocemos si es un parentesis de cierre
if(caracteractual==')')
{
	cuentatokens++;
	continue;
}
//Reconocemos si es una coma
if(caracteractual==',')
{
	cuentatokens++;
	continue;
}
//Reconocemos si es un punto y final
if(caracteractual=='.')
{
	cuentatokens++;
	continue;
}

//Reconocemos si es un punto y como
if(caracteractual==';')
{
	cuentatokens++;
	continue;
}
//Reconocemos si es un signo de suma
if(caracteractual=='+')
{
	cuentatokens++;
	continue;
```

```
}
//Reconocemos si es un signo de resta
if(caracteractual=='-')
{
	cuentatokens++;
	continue;
}
//Reconocemos si es un signo de multiplicación
if(caracteractual=='*')
{
	cuentatokens++;
	continue;
}
//Reconocemos si es un signo de división
if(caracteractual=='/')
{
	cuentatokens++;
	continue;
}
//Reconocemos si es un signo de operacion módulo
if(caracteractual=='|')
{
	cuentatokens++;
	continue;
}
//Reconocemos si es un signo de relación igual
if(caracteractual=='=')
{
	cuentatokens++;
	continue;
}
//Reconocemos si es el simbolo de asignación
if(caracteractual==':')
{
	temp=fgetc(ptrToFile);
	if(temp=='=')
		{
		cuentatokens++;
		continue;
		}
	ungetc(temp,ptrToFile);
	//Llamada a la función error los: no pueden ir solas
	continue;
}
//reconocemos si es un signo de realacion menor que, menor o igual
//que y desigualdad respectivamente
if(caracteractual=='<')
{
	temp=fgetc(ptrToFile);
	if(temp=='=')
```

```
		{
			cuentatokens++;
		}
		else if(temp=='>')
			{
			cuentatokens++;
			}
		else
		{
			ungetc(temp,ptrToFile);
			cuentatokens;
		}
		continue;
}
//Reconocemos si es un signo de relación mayor que o mayor o igual
if(caracteractual=='>')
{
		temp=fgetc(ptrToFile);
		if(temp=='=')
		{
			cuentatokens++;
		}
		else if(temp=='>')
			{
			cuentatokens++;
			}
		else
		{
			ungetc(temp,ptrToFile);
			cuentatokens;
		}
		continue;
}
//Procesamos los comentarios, no hay acción del scanner
if(caracteractual=='(')
{
		temp=fgetc(ptrToFile);
		while(temp!='}')
			temp=fgetc(ptrToFile);
}
else
{
		//mensaje de error en la construcción del comentario
		printf("Comentario mal construido\n");
}
//Reconocemos si hemos llegado al fin del archivo de lectura
if(!feof(ptrToFile))
{
		printf("Fin del scaneo.....\n");
}
```

```
        }//Fin del while principal del scanner
        //mandamos a imprimir lo solicitado en la práctica numero 2
        printf("Numero de lineas = %d\n",cuentalineas);
        printf("Numero de tokens = %d\n",cuentatokens);
        fclose(ptrToFile);
} //fin el main
```

SOLUCIÓN DE LA PRÁCTICA NO. 3

```
//Máxima longitud de los identificadores en SLANG
#define max_id_length 32
//redefinicion de char

typedef char string[max_id_length +1];
static string buffer;
static int valorNum=-1;
int flagid =0, flagnum=0, flagerror=0;
int numLin=1;

//definición de constante NULL para el tipo string..

const string nulo="\0";

typedef enum token_type
{
        OPEN_, CLOSE_, LIST_, PERIOD_, SEPARATOR_, ASSIGN_, PLUSOP_,
        MINUSOP_, TIMESOP_, OVEROP_, MODOP_, EQUAL_, NOTEQUAL_,
        LESS_, LESS_EQ_, GREAT_, GREAT_EQ_, BEGIN_, END_, ID_,
        INTEGER_, NUMBER_, VAR_, PROCEDURE_, CALL_, READ_, WRITE_,
        IF_, THEN_, ELSE_, FI_, WHILE_, DO_, OD_, NEG_, ABS_,
        SCANEOF_}tokens;

//Nombre de los tokens
        char *tokenName[] = {
        "OPEN_","CLOSE_", "LIST_", "PERIOD_", "SEPARATOR_", "ASSIGN_",
        "PLUSOP_", "MINUSOP_", "TIMESOP_", "OVEROP_", "MODOP_",
        "EQUAL_", "NOTEQUAL_", "LESS_", "LESS_EQ_", "GREAT_",
        "GREAT_EQ_", "BEGIN_", "END_", "ID_", "INTEGER_", "NUMBER_",
        "VAR_", "PROCEDURE_", "CALL_", "READ_", "WRITE_", "IF_",
        "THEN_", "ELSE_", "FI_", "WHILE_", "DO_", "OD_", "NEG_", "ABS_",
        "SCANEOF_"};


//Estructura de los nodos de la lista de los tokens
typedef struct listTok listaTokens;
struct listTok
{
        tokens t;
        string  n;
        int v;
        int nl;
        listaTokens *siguiente;
};

//Nuevo nodo
```

```
listaTokens *NuevoToken(void)
{
	listaTokens *a=(listaTokens *)malloc(sizeof(listaTokens));
	if(a==NULL) //si no hay memoria dispoible
	{
		printf("\n ERROR: No hay memoria");
		exit(0);
	}
	return(a);
}

//Función insertar en la lista de tokens...

void InsertarLT(listaTokens **inicio, tokens t)
{
	listaTokens *nuevo, *actual;
	nuevo = NuevoToken();
	nuevo->t=t;
	if(flagid==1)
	{
		strcpy(nuevo->n,buffer);
		flagid=0;
	}
	else
		strcpy(nuevo->n,nulo);

	if(flagnum==1)
	{
		nuevo->v=valorNum;
		flagnum=0;
	}
	else
		nuevo->v=1;

	nuevo->nl=numLin;
	nuevo->siguiente=NULL;
	actual=*inicio;
	//si la lista de tokens esta vacía..
	if(actual==NULL)
	{
		*inicio=nuevo;
		return;
	}

//si no esta vacía se mueve el puntero hasta el final.....
	while((actual->siguiente)!=NULL)
		actual=actual->siguiente;
	actual->siguiente=nuevo;
}
```

```
//Función que visualizara la lista completa de tokens..
void VisualizarLT(listaTokens *inicio)
{
	if(inicio==NULL)
	{
		printf("\n La lista esta vacia");
		return;
	}
	while(inicio!=NULL)
	{
		printf("\n\tLin# : %d\t",inicio->nl);
		printf("%s TOKENS \t",tokenName[inicio->t]);
		printf("\n");
		inicio=inicio->siguiente;
	}
}

//Función para borrar la lista de tokens de la memoria
void BorrarLT(listaTokens **inicio)
{
	listaTokens *temp;
	temp=*inicio;
	while(temp!=NULL)
	{
		*inicio=(*inicio)->siguiente;
		free(temp);
		temp=*inicio;
	}
}

**********************************************************************

#include<ctype.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include "lista.h"

FILE *ptrToFile;
listaTokens *inicioLT=NULL;

//Función para limpiar el buffer
void limpiar_buffer(void)
{
	int x;
	for(x=0; x<max_id_length; buffer[x++]='\0');
}

//Función para emitir los mensajes de errores léxicos..
```

```
void error_lexico(int nl, char a)
{
	printf("ERROR(linea: %d): Caracter Desconocido->%c\n",nl,a);
}

tokens chequear_reservada(void)
{
	if(strcmp(strlwr(buffer),"begin")==0)
		return(BEGIN_);
	if(strcmp(strlwr(buffer),"end")==0)
		return(END_);
	if(strcmp(strlwr(buffer),"int")==0)
		return(INTEGER_);
	if(strcmp(strlwr(buffer),"var")==0)
		return(VAR_);
	if(strcmp(strlwr(buffer),"procedure")==0)
		return(PROCEDURE_);
	if(strcmp(strlwr(buffer),"call")==0)
		return(CALL_);
	if(strcmp(strlwr(buffer),"read")==0)
		return(READ_);
	if(strcmp(strlwr(buffer),"write")==0)
		return(WRITE_);
	if(strcmp(strlwr(buffer),"if")==0)
		return(IF_);
	if(strcmp(strlwr(buffer),"then")==0)
		return(THEN_);
	if(strcmp(strlwr(buffer),"else")==0)
		return(ELSE_);
	if(strcmp(strlwr(buffer),"fi")==0)
		return(FI_);
	if(strcmp(strlwr(buffer),"while")==0)
		return(WHILE_);
	if(strcmp(strlwr(buffer),"do")==0)
		return(DO_);
	if(strcmp(strlwr(buffer),"od")==0)
		return(OD_);
	if(strcmp(strlwr(buffer),"neg")==0)
		return(NEG_);
	if(strcmp(strlwr(buffer),"abs")==0)
		return(ABS_);
	return(ID_);
}

//scanner

tokens scanner(void)
{
	tokens ResTemp;
	char input, c;
```

```
int i=0;

while((input=fgetc(ptrToFile))!=EOF)
{
	i=0;
	limpiar_buffer();
	if(input=='\n')
		numLin++;
	else if(isspace(input))
		continue;
	else if(isalpha(input))
	{
		buffer[i++]=input;
		while(isalnum(c=fgetc(ptrToFile)))
			buffer[i++]=c;
		ungetc(c,ptrToFile);
		ResTemp=chequear_reservada();
		if(ResTemp==ID_)
			flagid =1;
		return(ResTemp);
	}

	else if(isdigit(input))
	{
		buffer[i++]=input;
		while(isdigit(c=fgetc(ptrToFile)))
			buffer[i++]=c;
		valorNum=atoi(buffer);
		flagid=1;
		ungetc(c,ptrToFile);
		return(NUMBER_);
	}
	else if(input=='(')
		return(OPEN_);
	else if(input==')')
		return(CLOSE_);
	else if(input==',')
		return(LIST_);
	else if(input=='.')
		return(PERIOD_);
	else if(input==';')
		return(SEPARATOR_);
	else if(input==':')
	{
		c=fgetc(ptrToFile);
	if(c=='=')
		return(ASSIGN_);
	else
	{
		ungetc(c,ptrToFile);
```

```
                error_lexico(numLin,input);
                flagerror=1;
                }
        }

        else if(input=='+')
                return(PLUSOP_);
        else if(input=='-')
                return(MINUSOP_);
        else if(input=='*')
                return(TIMESOP_);
        else if(input=='/')
                return(OVEROP_);
        else if(input=='|')
                return(MODOP_);
        else if(input=='=')
                return(EQUAL_);
        else if(input=='<')
        {
                c=fgetc(ptrToFile);
                if(c=='>')
                        return(NOTEQUAL_);
                else if(c=='=')
                        return(LESS_EQ_);
                else
                {
                        ungetc(c,ptrToFile);
                        return(LESS_);
                }
        }

        else if(input=='>')
        {
                c=fgetc(ptrToFile);
                if(c=='=')
                        return(GREAT_EQ_);
                else if(c=='=')
                        return(LESS_EQ_);
                else
                {
                        ungetc(c,ptrToFile);
                        return(GREAT_);
                }
        }

//Para procesar comentarios
else if(input=='{')
{
        c=fgetc(ptrToFile);
        while(c!='}')
```

```
                c=fgetc(ptrToFile);

}
else    //en cualquier otro caso
{
        error_lexico(numLin,input);
        flagerror=1;
}
}

if(feof(ptrToFile))
        return(SCANEOF_);
}


**************************************************************************

#include"scan.h"

main()
{
        tokens t;
        char NombreArchivo[12];

        printf("\nNombre del archivo");
        gets(NombreArchivo);

        if((ptrToFile=fopen(NombreArchivo,"r"))==NULL)
        {
                printf("\n El archivo no se puede abrir\n");
                exit(0);
        }
        while(!ferror(ptrToFile)&&!feof(ptrToFile))
        {
                t=scanner();
                if(flagerror==0)
                        InsertarLT(&inicioLT,t);
                flagerror=0;
        }
        fclose(ptrToFile);
        VisualizarLT(inicioLT);
}
```

SOLUCIÓN DE LA PRÁCTICA NO. 4

********************************Lista.h********************************

```
typedef enum tokens_type
{
        OPEN_, CLOSE_, LIST_, PERIOD_, SEPARATOR_,ASSIGN_, PLUSOP_,
        MINUSOP_, TIMESOP_, OVEROP_, MODOP_, EQUAL_, NOTEQUAL_,
        LESS_, LESS_EQUAL_, GREAT_, GREAT_EQ_, BEGIN_, END_, ID_,
        INTERGER_, NUMBER_, VAR_, PROC_, CALL_, READ_, WRITE_, IF_,
        THEN_, ELSE_, FI_, WHILE_, DO_, OD_, NEGATE_, ABS_
}tokens;

typedef struct lt
{
        tokens t;
        char nombre;
        int n;
        struct lt *sig;
}listTokens;
void insertar(listTokens **cablt,tokens t,int n);
void visualizar(listTokens **cab);
void borrar(listTokens **cab);
listTokens *NuevoElemento();

char buffer[32];
char s[32];
char nu[32];
char *cad;
FILE *arch;
int numlin=1;
tokens restemp;
int error;
int ctokens=0;
int var=0,j=0,k=0,p=0;
int simb=0;
char *tokenName[]={
        "OPEN", "CLOSE", "LIST", "PERIOD", "SEPARATOR","ASSIGN",
        "PLUSOP", "MINUSOP", "TIMESOP", "OVEROP", "MODOP", "EQUAL",
        "NOTEQUAL", "LESS", "LESS_EQUAL", "GREAT", "GREAT_EQ",
        "BEGIN", "END", "ID", "INTERGER", "NUMBER", "VAR", "PROCEDURE",
        "CALL", "READ", "WRITE", "IF", "THEN", "ELSE", "FI", "WHILE", "DO",
        "OD", "NEGATE", "ABS"};

void limpiarbuffer();
tokens chequear_reservada();
void lexical_error(int nlinea,char caracter);
tokens scanner();
```

**********Scan.h**********

```
listTokens *NuevoElemento()
{
	return ((listTokens *)malloc(sizeof(listTokens)));
}
void insertar(listTokens **cab,tokens t,int n)
{
	listTokens *cabecera=*cab;
	listTokens *actual=cabecera,*anterior=cabecera,*q;
	if(cabecera==NULL)
	{
		cabecera=NuevoElemento();
		cabecera->t=t;
		cabecera->n=n;
		cabecera->sig=NULL;
		*cab=cabecera;
		return;
	}
	while(actual!=NULL )
	{
		anterior=actual;
		actual=actual->sig;
	}
	q=NuevoElemento();
	q->t=t;
	q->n=n;
	q->sig=actual;
	anterior->sig=q;
	*cab=cabecera;
}
void visualizar(listTokens **cab)
{
	listTokens*actual=*cab;
	int i=0,k=0;
	if(cab==NULL)
		printf("Lista Vacia\n");
	else
	{
		system("pause");
		printf("\tNumero de linea:%d\n",actual->n);
		while(actual!=NULL)
		{
			if(actual->t==BEGIN_)
			{
				printf("El nombre del tokens: %s\n",tokenName[actual-
>t]);
				/*printf("Lin#:%d		%s_ TOKENS\n",actual-
>n,tokenName[actual->t]);
				system("pause");*/
```

```
                        }
                        else if(actual->t==ID_)
                        {
                                printf("El nombre del tokens: %s\t El nombre del
identificador:%c\n",tokenName[actual->t],s[i++]);
                                //printf("Lin#:%d          %s_ TOKENS\t El nombre
del identificador:%c\n",actual->n,tokenName[actual->t],s[i++]);
                        }
                        else if(actual->t==NUMBER_)
                        {
                                printf("El nombre del tokens: %s\t Su valor es
=>%c\n",tokenName[actual->t],nu[k++]);
                                //printf("Lin#:%d          %s_ TOKENS\t Su valor es
=>%c\n",actual->n,tokenName[actual->t],nu[k++]);
                        }
                        else if(actual->t==EQUAL_)
                        {
                                actual=actual->sig;
                        }
                        else
                        {
                                printf("El nombre del tokens: %s\n",tokenName[actual-
>t]);
                                ///printf("Lin#:%d          %s_ TOKENS\n",actual-
>n,tokenName[actual->t]);
                        }
                        if(actual->n<6)
                        {
                                if(actual->n!=actual->sig->n)
                                {
                                        system("pause");
                                        printf("\tNumero de linea:%d\n",actual->n+1);
                                }
                        }
                        actual=actual->sig;

                }
        }
}
tokens scanner(void)
{
        char input,c;
        int i=0,aux;
        while((input=fgetc(arch))!=EOF)
        {
                limpiarbuffer();
                if(input=='\n')
                        numlin++;
                if(isspace(input))
                        continue;
```

```
else if(isalpha(input))
{
	buffer[i++]=input;
	for(c=fgetc(arch);isalnum(c);c=fgetc(arch))
	{
		buffer[i++]=c;
	}
	ungetc(c,arch);
	restemp=chequear_reservada();
	if(restemp==ID_)
	{
		s[j++]=input;
	}
	break;
}
else if(input=='.')
{
	restemp=PERIOD_;
	break;
}
else if(input=='+')
{
	restemp=PLUSOP_;
	break;
}
else if(input=='<')
{
	aux=ftell(arch);
	input=fgetc(arch);
	if(input=='>')
	{
		restemp=NOTEQUAL_;
	}
	else
	{
		restemp=LESS_;
	}
	fseek(arch,0,aux);
	break;
}
else if(input==':')
{
	aux=ftell(arch);
	input=fgetc(arch);
	if(input=='=')
	{
		restemp=ASSIGN_;
	}
	fseek(arch,0,aux);
	break;
```

```
		}
		else if(input=='=')
		{
			restemp=EQUAL_;
			break;
		}
		else if(input=='(')
		{
			restemp=OPEN_;
			break;
		}
		else if(input==')')
		{
			restemp=CLOSE_;
			break;
		}
		else if(input==';')
		{
			restemp=SEPARATOR_;
			break;
		}
		else if(input==',')
		{
			restemp=LIST_;
			break;
		}
		else if(isdigit(input))
		{
			buffer[i++]=input;
			while(isdigit((c=fgetc(arch))))
				buffer[i++]=c;
			ungetc(c,arch);
			restemp=NUMBER_;
			nu[k++]=input;
			break;
		}
		/*else
		{
			lexical_error(numlin,input);
		}*/
	}
	return(restemp);
}
void limpiarbuffer()
{
	int i;
	for(i=0;i<32;i++)
	{
		buffer[i]='\0';
	}
```

```
}
void lexical_error(int nlinea,char caracter)
{
	printf("ERROR(linea:%d): Caracter Desconocido->%c \n",nlinea,caracter);
}
tokens chequear_reservada()
{
	int i=0;
	while(tokenName[i]!=NULL)
	{
		if(!(strcmpi(buffer,tokenName[i])))
		{
			restemp=i;
			break;

		}
		else if(!(strcmpi(buffer,"INT")))
		{
			restemp=INTERGER_;
		}
		else
		{
			restemp=ID_;
		}
		i++;
	}
	return (restemp);
}
```

******************************Parser.h*******************************

```
void program_declaration(void);
void block(void);
void declaration_part(void);
void constant_declaration(void);
void procedure_declaration(void);
void assign();
void statement_part(void);
void match(tokens);
void sintaxis_error(tokens t, int l);
tokens get_token();
/*******************VARIABLES
GLOBALES*********************************/
listTokens *token_actual;
listTokens *cablt2;
tokens temp;
/*****************************************************************/
void variable_declaration();
void PROC_declaration();
void type_declarer();
```

```
void identifier();
void number();

void sintaxis_error(tokens t, int l)
{
	if(t==ID_)
		printf("Error de sintaxis:%s Linea: %d, token esperado SEPARATOR_,
o LIST_ \n",tokenName[t],l-1);
	else
		printf("Error de sintaxis:%s Linea: %d\n",tokenName[t],l);
}
void match(tokens t)
{
	tokens temptoken;
	temptoken=get_token();
	if(temp==t)
	{
		token_actual=token_actual->sig;
		return;
	}
	else
	{

		sintaxis_error(temptoken,token_actual->n);
	}

}
tokens get_token()
{
	temp=token_actual->t;
	return(temp);
}
void program_declaration(void)
{
	token_actual=cablt2;
	block();
	match(PERIOD_);
}
void block(void)
{
	match(BEGIN_);
	declaration_part();
	statement_part();
	match(END_);
}
void assign()
{
	tokens temptoken;
	int aux=0;
	temptoken=get_token();
```

```
	if(temptoken==ASSIGN_)
	{
		while(temptoken==ASSIGN_)
		{
			match(ASSIGN_);
			number();
			temptoken=token_actual->t;
		}
	}
	else if(temptoken==EQUAL_)
	{
		while(temptoken==EQUAL_)
		{
			match(EQUAL_);
			sintaxis_error(temptoken,token_actual->n);
			identifier();
			aux=token_actual->t;
			if(aux==PLUSOP_)
				match(PLUSOP_);
			else if(aux==MINUSOP_)
				match(MINUSOP_);
			else if(aux==MODOP_)
				match(MODOP_);
			identifier();
			temptoken=token_actual->t;
		}
	}
}
void statement_part()
{
	tokens temptoken=get_token();
	while(temptoken==ID_)
	{
		identifier();
		assign();
		match(SEPARATOR_);
		temptoken=token_actual->t;
	}

}
void declaration_part(void)
{
	int control=0;
	tokens temptoken=get_token();
	while((temptoken==INTERGER_)||(temptoken==VAR_))
	{
		if(temptoken==INTERGER_)
			constant_declaration();
		else if(temptoken==VAR_)
			variable_declaration();
```

```
                control=1;
                temptoken=get_token();
        }
        if(control==1)
        {
                temptoken=get_token();
                while(temptoken==PROC_)
                {
                        PROC_declaration();
                }
        }
        else if(control==0)
        {
                while(temptoken==PROC_)
                {
                        PROC_declaration();
                }
        }
}
void constant_declaration()/***************CONSTANT*****************/
{
        tokens temptoken;
        type_declarer();
        identifier();
        match(EQUAL_);
        number();
        temptoken=get_token();
        while(temptoken==LIST_)
        {
                match(LIST_);
                identifier();
                match(EQUAL_);
                number();
        }
        match(SEPARATOR_);
}
void
identifier()/************************IDENTIFIER*****************************
***/
{
        match(ID_);
}
void
number()/*********************NUMBER*********************************
**/
{
        match(NUMBER_);
}
void type_declarer()
{
```

```
	match(INTERGER_);
}
void variable_declaration()
{
	tokens temptoken;
	match(VAR_);
	type_declarer();
	identifier();
	temptoken=get_token();
	while(temptoken==LIST_)
	{
		match(LIST_);
		identifier();
		temptoken=token_actual->t;
	}
	match(SEPARATOR_);

}
void PROC_declaration()
{
	match(PROC_);
	identifier();
	block();
	match(SEPARATOR_);
}
/*********************************Parser.c*********************************/
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<string.h>
#include "lista.h"
#include "scan.h"
#include "parser.h"

int main()
{
	char listar,nombre[12];
	tokens t;
	listTokens *cablt=NULL;
	system("cls");
	puts("Nombre del archivo fuente");
	gets(nombre);
	if((arch=fopen(nombre,"r"))==NULL)
	{
		puts("El archivo no se pudo abrir");
		exit(0);
	}
	else if(!ferror(arch)&&!feof(arch))
	{
		while(!ferror(arch) && !feof(arch))
```

```
		{
			t=scanner();
			insertar(&cablt,t,numlin);
			if(t==PERIOD_)
				break;
		}
	}
	printf("Fin del scanneo\n");
	printf("Desea visualizar la lista de tokens(S/N)\n");
	scanf("%c",&listar);
	if(listar=='s'||listar=='S')
	{
		visualizar(&cablt);
	}
	cablt2=cablt;
	puts("\nErrores Encontrados:");
	program_declaration();
	puts("**Fin del parseo**");
	fclose(arch);
	return 0;
}
```

SOLUCIÓN DE LA PRÁCTICA NO. 5

Parser.h

```
#include <stdio.h>

void syntaxis_error(tokens temptoken,int numlin);
tokens get_token(void);
void block(void);
void declaration_part(void);
void constant_declaration(void);
void type_declarer(void);
void variable_declaration(void);
void procedure_declaration(void);
void statement_part(void);
void statement(void);
void else_part(void);
void expresion(void);
void term(void);
void unary_op(void);
void relation(void);
void relational_operator(void);
void identifier(void);
void number(void);

int kase2=0;
lt *token_actual=NULL;/*token_actual  verifica cada terminal en el lado derecho de
una produccion
                es el simbolo al cual se esta apuntando actualmente en la lista de tokens
                que produjo el scanner*/
tokens temptoken;
int err=0;

void match(tokens t)
{
   temptoken=get_token();
     if(temptoken==t)
     {
        token_actual=token_actual->siguiente;
        return;
     }
     else
        syntaxis_error(t,token_actual->nl);
}

void syntaxis_error(tokens temptoken,int numlin)
{
   if(kase2==0)
```

```
   printf("\n\tERROR en la linea#: %d, Se esperaba:
%s",numlin,tokenName[temptoken]);
   err=1;
   total_err_parser++;
}

tokens get_token(void)
{
   temptoken=token_actual->t;
   return(temptoken);
}

void program_declaration(void)
{
   token_actual=cab;
   block();
   match(PERIOD_);
}

void block(void)
{
   int e=0;
   match(BEGIN_);
   temptoken=get_token();
     if(temptoken==END_ || temptoken==SCANEOF_ || temptoken==PERIOD_)
        goto e;
   declaration_part();
   temptoken=get_token();
     if(temptoken==END_ || temptoken==SCANEOF_ || temptoken==PERIOD_)
        goto e;
   statement_part();
   e:
   match(END_);
}

void declaration_part(void)
{
   int control=0;
   temptoken=get_token();
     while(temptoken==INTEGER_ || temptoken==VAR_)
     {
        if(temptoken==INTEGER_)
           constant_declaration();
        else if(temptoken==VAR_)
           variable_declaration();
        control=1;
        temptoken=get_token();
     }
     if(control==1)
     {
```

```
        temptoken=get_token();
          while(temptoken==PROCEDURE_)
          {
             procedure_declaration();
             temptoken=get_token();
          }
     }
     else if(control==0)
     {
        while(temptoken==PROCEDURE_)
        {
           procedure_declaration();
           temptoken=get_token();
        }
     }
}

void constant_declaration(void)
{
   type_declarer();
   identifier();
   match(EQUAL_);
   number();
   temptoken=get_token();
     while(temptoken==LIST_)
     {
        match(LIST_);
        identifier();
        match(EQUAL_);
        number();
        temptoken=get_token();
     }
   match(SEPARATOR_);
}

void type_declarer(void)
{
   match(INTEGER_);
}

void variable_declaration(void)
{
   match(VAR_);
   type_declarer();
   identifier();
   temptoken=get_token();
     while(temptoken==LIST_)
     {
        match(LIST_);
        identifier();
```

```
        temptoken=get_token();
     }
   match(SEPARATOR_);
}

void procedure_declaration(void)
{
   match(PROCEDURE_);
   identifier();
   block();
   match(SEPARATOR_);
}

void statement_part(void)
{
   statement();
   match(SEPARATOR_);
   temptoken=get_token();
     while(temptoken==ID_ || temptoken==IF_ || temptoken==WHILE_ ||
temptoken==CALL_ || temptoken==READ_ || temptoken==WRITE_)
     {
        statement();
        match(SEPARATOR_);
        temptoken=get_token();
     }
}

void statement(void)
{
   temptoken=get_token();
     while(temptoken==ID_ || temptoken==IF_ || temptoken==WHILE_ ||
temptoken==CALL_ || temptoken==READ_ || temptoken==WRITE_)
     {
        if(temptoken==ID_)
        {
           identifier();
           match(ASSIGN_);
           temptoken=get_token();
             if(temptoken==OPEN_ || temptoken==CLOSE_ || temptoken==LIST_ ||
temptoken==PERIOD_ || temptoken==SEPARATOR_ || temptoken==PLUSOP_ ||
temptoken==MINUSOP_ || temptoken==TIMESOP_ || temptoken==OVEROP_ ||
temptoken==MODOP_ || temptoken==EQUAL_ || temptoken==NOTEQUAL_ ||
temptoken==LESS_ || temptoken==LESS_EQ_ || temptoken==GREAT_ ||
temptoken==GREAT_EQ_)
                token_actual=token_actual->siguiente;
           expresion();
        }
        else if(temptoken==IF_)
        {
           match(IF_);
```

```
            relation();
            match(THEN_);
            statement_part();
            else_part();
            match(FI_);
            temptoken=get_token();
              if((temptoken!=SEPARATOR_) && (temptoken==ID_ || temptoken==IF_ ||
temptoken==WHILE_ || temptoken==CALL_ || temptoken==READ_ ||
temptoken==WRITE_))
                  match(SEPARATOR_);
         }
         else if(temptoken==WHILE_)
         {
            match(WHILE_);
            relation();
            match(DO_);
            statement_part();
            match(OD_);
            temptoken=get_token();
              if((temptoken!=SEPARATOR_) && (temptoken==ID_ || temptoken==IF_ ||
temptoken==WHILE_ || temptoken==CALL_ || temptoken==READ_ ||
temptoken==WRITE_))
                  match(SEPARATOR_);
         }
         else if(temptoken==CALL_)
         {
            match(CALL_);
            identifier();
         }
         else if(temptoken==READ_)
         {
            match(READ_);
            match(OPEN_);
            identifier();
            temptoken=get_token();
              while(temptoken==LIST_)
              {
                 identifier();
                 temptoken=get_token();
              }
            match(CLOSE_);
            temptoken=get_token();
              if((temptoken!=SEPARATOR_) && (temptoken==ID_ || temptoken==IF_ ||
temptoken==WHILE_ || temptoken==CALL_ || temptoken==READ_ ||
temptoken==WRITE_))
                  match(SEPARATOR_);
         }
         else if(temptoken==WRITE_)
         {
            match(WRITE_);
```

```
            match(OPEN_);
            expresion();
            temptoken=get_token();
              while(temptoken==LIST_)
              {
                 expresion();
                 temptoken=get_token();
              }
            match(CLOSE_);
            temptoken=get_token();
              if((temptoken!=SEPARATOR_) && (temptoken==ID_ || temptoken==IF_ ||
temptoken==WHILE_ || temptoken==CALL_ || temptoken==READ_ ||
temptoken==WRITE_))
                 match(SEPARATOR_);
        }
        temptoken=get_token();
     }
}

void else_part(void)
{
   match(ELSE_);
   statement_part();
}

void expresion(void)
{
   unary_op();
   term();
   temptoken=get_token();
     if(temptoken!=SEPARATOR_ && temptoken!=PLUSOP_ &&
temptoken!=MINUSOP_ && temptoken!=TIMESOP_ && temptoken!=OVEROP_
&& temptoken!=MODOP_)
     {
        if(temptoken==ID_)
        {
           syntaxis_error(OPERATOR_,token_actual->nl);
           identifier();
        }
        if(temptoken==NUMBER_)
        {
           syntaxis_error(OPERATOR_,token_actual->nl);
           number();
        }
     }
     while(temptoken==PLUSOP_ || temptoken==MINUSOP_ ||
temptoken==TIMESOP_ || temptoken==OVEROP_ || temptoken==MODOP_)
     {
        if(temptoken==PLUSOP_)
        {
```

```
            match(PLUSOP_);
            term();
         }
         else if(temptoken==MINUSOP_)
         {
            match(MINUSOP_);
            term();
         }
         else if(temptoken==TIMESOP_)
         {
            match(TIMESOP_);
            term();
         }
         else if(temptoken==OVEROP_)
         {
            match(OVEROP_);
            term();
         }
         else if(temptoken==MODOP_)
         {
            match(MODOP_);
            term();
         }
         temptoken=get_token();
      }
}

void term(void)
{
   temptoken=get_token();
     if(temptoken==OPEN_)
     {
        match(OPEN_);
        expresion();
        match(CLOSE_);
     }
     if(temptoken==ID_)
        identifier();
     if(temptoken==NUMBER_)
        number();
}

void unary_op(void)
{
   temptoken=get_token();
     if(temptoken==NEGATE_)
        match(NEGATE_);
     if(temptoken==ABS_)
        match(ABS_);
}
```

```
void relation(void)
{
   match(OPEN_);
   expresion();
   relational_operator();
   expresion();
   match(CLOSE_);
}

void relational_operator(void)
{
   temptoken=get_token();
     if(temptoken==EQUAL_)
        match(EQUAL_);
     if(temptoken==NOTEQUAL_)
        match(NOTEQUAL_);
     if(temptoken==LESS_)
        match(LESS_);
     if(temptoken==LESS_EQ_)
        match(LESS_EQ_);
     if(temptoken==GREAT_)
        match(GREAT_);
     if(temptoken==GREAT_EQ_)
        match(GREAT_EQ_);
}

void identifier(void)
{
   match(ID_);
}

void number(void)
{
   match(NUMBER_);
}
```

Scanner.h

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include "ListaT.h"

/*------------------Definiciones de las funciones de la lista-----------------*/
lt *nuevoelemento(void);//Crea el bloque de memoria
void insertar_tokens_en_lista(lt **list,tokens to);//Inserta los elementos de la lista
void visualizar(lt *list);//Visualiza la lista
void liberar(lt **list);//Libera la memoria
/*-----------------Definiciones de las funciones del scanner-----------------*/
```

```
tokens scanner(void);//Regresa la clasificacion del token
void limpiar_buffer(void);//Pone a cero el buffer
tokens chequear_reservada(void);//Verifica de que tipo es el token
void lexical_error(int nl,char ipt);//Imprime mensajes de errores

/*----------------------Declaraciones de las variables----------------------*/
static string buffer;//Variable char buffer[32] de tipo static
int flagid=0,flagnum=0,flagerror=0,flagEOF=0;//Banderas
int numlin=1;
lt *cab=NULL;
const string nulo="\0";//Variable char nulo[]="\0"
tokens restemp;//Variable de tipo enumerado tokens
FILE *ptrtofile;
int
total_id=0,total_err_scan=0,total_err_parser=0,total_pal_reser=0,total_token=0,total_si
m_espe=0;

/*----------------Declaraciones de las funciones de la lista------------------*/
lt *nuevoelemento(void)
{
   lt *temp=(lt *)malloc(sizeof(lt));
     if(!(temp))
     {
        printf("\n\tERROR: Insuficiente espacio de memoria");
        printf("\n\tSaliendo del programa...");
        system("pause");
        exit(-1);
     }
   return(temp);
}

void insertar_tokens_en_lista(lt **list,tokens to)
{
   lt *temp=NULL,*aux=*list,*aux2=*list;
     if(aux==NULL)
     {
        temp=nuevoelemento();
        total_token++;
        temp->t=to;
          if(flagid==1)
          {
             strcpy(temp->n,buffer);
             flagid=0;
          }
          else
             memset(temp->n,0,sizeof(temp->n));
          if(flagnum==1)
          {
             temp->v=atoi(buffer);
             flagnum=0;
```

```
          }
          else
             temp->v=0;
        temp->nl=numlin;
        temp->siguiente=NULL;
        *list=aux=temp;
     }
     else
     {
        while(aux->siguiente!=NULL)
           aux=aux->siguiente;
        if(aux->siguiente==NULL)
        {
           temp=nuevoelemento();
           total_token++;
           temp->t=to;
           temp->nl=numlin;
            if(flagid==1)
            {
               strcpy(temp->n,buffer);
               flagid=0;
            }
            else
               memset(temp->n,0,sizeof(temp->n));
            if(flagnum==1)
            {
               temp->v=atoi(buffer);
               flagnum=0;
            }
            else
               temp->v=0;
           temp->siguiente=aux->siguiente;
           aux->siguiente=temp;
           *list=aux2;
        }
     }
}

void visualizar(lt *list)
{
   if(list==NULL)
   {
      printf("\n\tLa lista esta vacia...");
   }
   else
   {
      while(list!=NULL)
      {
         if(strcmp(tokenName[list->t],"ID_")==0)
```

```
            printf("\n\tLinea#: %d\t%sTOKEN\tNombre: %s",list->nl,tokenName[list-
>t],list->n);
          else if(strcmp(tokenName[list->t],"NUMBER_")==0)
            printf("\n\tLinea#: %d\t%sTOKEN\tNombre: %d",list->nl,tokenName[list-
>t],list->v);
          else
            printf("\n\tLinea#: %d\t%sTOKEN",list->nl,tokenName[list->t]);
          list=list->siguiente;
       }
    }
}

void liberar(lt **list)
{
   lt *temp=*list,*aux=NULL;
     while(temp!=NULL)
     {
        aux=temp->siguiente;
        free(temp);
        temp=aux;
     }
   *list=temp;
}

/*-----------------Declaraciones de las funciones del Scanner-----------------*/
tokens scanner (void)
{
  char input,c;
  int i=0;
  input=fgetc(ptrtofile);
    while(input!=EOF)
    {
       limpiar_buffer();
        if(input=='\n')
        {
           input=fgetc(ptrtofile);
           numlin++;
        }
        else if(isspace(input))
        {
           input=fgetc(ptrtofile);
           continue;
        }
        else if(isalpha(input))//a-z o A-Z
        {
           buffer[i++]=tolower(input);//Introducir el caracter en minuscula
             for(c=fgetc(ptrtofile);isalnum(c) && !feof(ptrtofile);c=fgetc(ptrtofile))//a-z o
A-Z o 0-9
                buffer[i++]=tolower(c);
             if(c==EOF)
```

```
        flagEOF=1;
   ungetc(c,ptrtofile);
   restemp=chequear_reservada();
   return restemp;
}
else if(isdigit(input))
{
   buffer[i++]=input;
     while(isdigit(c=fgetc(ptrtofile)) && !feof(ptrtofile))
        buffer[i++]=c;
     if(c==EOF)
        flagEOF=1;
   ungetc(c,ptrtofile);
   restemp=NUMBER_;
   flagnum=1;
   return restemp;
}
else if(ispunct(input))
{
     if(input=='(')
     {
        total_sim_espe++;
        return(OPEN_);
     }
     else if(input=='.')
     {
        total_sim_espe++;
        return(PERIOD_);
     }
     else if(input=='-')
     {
        total_sim_espe++;
        return(MINUSOP_);
     }
     else if(input==')')
     {
        total_sim_espe++;
        return(CLOSE_);
     }
     else if(input=='/')
     {
        total_sim_espe++;
        return(OVEROP_);
     }
     else if(input=='|')
     {
        total_sim_espe++;
        return(MODOP_);
     }
     else if(input=='*')
```

```
            {
               total_sim_espe++;
               return(TIMESOP_);
            }
            else if(input==',')
            {
               total_sim_espe++;
               return(LIST_);
            }
            else if(input==';')
            {
               total_sim_espe++;
               return(SEPARATOR_);
            }
            else if(input=='=')
            {
               total_sim_espe++;
               return(EQUAL_);
            }
            else if(input=='+')
            {
               total_sim_espe++;
               return(PLUSOP_);
            }
            else if(input=='&' || input=='#' || input=='$' || input=='?' || input=='^' ||
input=='~' || input=='@' || input=='%')
            {
               lexical_error(numlin,input);
               input=fgetc(ptrtofile);
               continue;
            }
            else
            {
            buffer[i++]=input;
              while(ispunct(c=fgetc(ptrtofile)) && !feof(ptrtofile))
                 buffer[i++]=c;
              if(c==EOF)
                 flagEOF=1;
            ungetc(c,ptrtofile);
            restemp=chequear_reservada();
            return restemp;
            }
         }
         else
         {
            lexical_error(numlin,input);
            input=fgetc(ptrtofile);
         }
      }//Fin del while
      if(input==EOF)
```

```
     {
        restemp=SCANEOF_;
        return restemp;
     }
}//fin de la funcion

void limpiar_buffer(void)
{
   strcpy(buffer,nulo);
   memset(buffer,0,sizeof(buffer));
}

tokens chequear_reservada(void)
{
   if(strcmp(buffer,"begin")==0)
   {
      total_pal_reser++;
      return(BEGIN_);
   }
   else if(strcmp(buffer,"var")==0)
   {
      total_pal_reser++;
      return(VAR_);
   }
   else if(strcmp(buffer,"int")==0)
   {
      total_pal_reser++;
      return(INTEGER_);
   }
   else if(strcmp(buffer,"procedure")==0)
   {
      total_pal_reser++;
      return(PROCEDURE_);
   }
   else if(strcmp(buffer,"while")==0)
   {
      total_pal_reser++;
      return(WHILE_);
   }
   else if(strcmp(buffer,"<>")==0)
   {
      total_sim_espe++;
      return(NOTEQUAL_);
   }
   else if(strcmp(buffer,"do")==0)
   {
      total_pal_reser++;
      return(DO_);
   }
   else if(strcmp(buffer,"read")==0)
```

```
{
   total_pal_reser++;
   return(READ_);
}
else if(strcmp(buffer,"od")==0)
{
   total_pal_reser++;
   return(OD_);
}
else if(strcmp(buffer,"end")==0)
{
   total_pal_reser++;
   return(END_);
}
else if(strcmp(buffer,":=")==0)
{
   total_sim_espe++;
   return(ASSIGN_);
}
else if(strcmp(buffer,"call")==0)
{
   total_pal_reser++;
   return(CALL_);
}
else if(strcmp(buffer,"write")==0)
{
   total_pal_reser++;
   return(WRITE_);
}
else if(strcmp(buffer,"if")==0)
{
   total_pal_reser++;
   return(IF_);
}
else if(strcmp(buffer,">=")==0)
{
   total_sim_espe++;
   return(GREAT_EQ_);
}
else if(strcmp(buffer,"<=")==0)
{
   total_sim_espe++;
   return(LESS_EQ_);
}
else if(strcmp(buffer,"<")==0)
{
   total_sim_espe++;
   return(LESS_);
}
else if(strcmp(buffer,">")==0)
```

```
    {
       total_sim_espe++;
       return(GREAT_);
    }
    else if(strcmp(buffer,"then")==0)
    {
       total_pal_reser++;
       return(THEN_);
    }
    else if(strcmp(buffer,"else")==0)
    {
       total_pal_reser++;
       return(ELSE_);
    }
    else if(strcmp(buffer,"fi")==0)
    {
       total_pal_reser++;
       return(FI_);
    }
    else if(strcmp(buffer,"neg")==0)
    {
       total_pal_reser++;
       return(NEGATE_);
    }
    else if(strcmp(buffer,"abs")==0)
    {
       total_pal_reser++;
       return(ABS_);
    }
    else
    {
       flagid=1;
       total_id++;
       return(ID_);
    }
}

void lexical_error(int nl,char ipt)
{
   flagerror=1;
   total_err_scan++;
   printf("\n\tERROR: En Linea: %d, Caracter Desconocido: %c",nl,ipt);
}
```

ListaT.h

```
#define max_id_length 32//Longuitud maxima de un token

typedef char string[max_id_length+1];//Sobre_nombre de char string[32]
```

```
typedef enum token_type//Representa el cjto de tokens del SLANG
{
OPEN_, CLOSE_, LIST_, PERIOD_, SEPARATOR_, ASSIGN_, PLUSOP_,
MINUSOP_,
TIMESOP_, OVEROP_, MODOP_, EQUAL_, NOTEQUAL_, LESS_, LESS_EQ_,
GREAT_,
GREAT_EQ_, BEGIN_, END_, ID_, INTEGER_, NUMBER_, VAR_,
PROCEDURE_, CALL_,
READ_, WRITE_, IF_, THEN_, ELSE_, FI_, WHILE_, DO_, OD_, NEGATE_,
ABS_, SCANEOF_, OPERATOR_
}tokens;

char *tokenName[]=//Para hacer uso de los nombres de los tokens en el
{               //contexto de las lineas de codigo
"OPEN_","CLOSE_","LIST_","PERIOD_","SEPARATOR_","ASSIGN_","PLUSOP_"
,"MINUSOP_",
"TIMESOP_","OVEROP_","MODOP_","EQUAL_","NOTEQUAL_","LESS_","LESS
_EQ_","GREAT_",
"GREAT_EQ_","BEGIN_","END_","ID_","INTEGER_","NUMBER_","VAR_","PRO
CEDURE_","CALL_",
"READ_","WRITE_","IF_","THEN_","ELSE_","FI_","WHILE_","DO_","OD_","NEG
ATE_","ABS_","SCANEOF_","OPERATOR_"
};

typedef struct list_tok lt;
struct list_tok
{
   tokens t;
   string n;//El nombre de la variable
   int v;//Valor del Tipo reservado number
   int nl;//Valor del numero de linea
   lt *siguiente;
};
```

cnjtprimero.h

```
#include<stdio.h>

/******************DECLARACION DE LAS
FUNCIONES*******************/
void cpprogram_declaration(void);
void cpblock(void);
void cpdeclaration_part(void);
void cpconstant_declaration(void);
void cptype_declarer(void);
void cpvariable_declaration(void);
void cpproc_declaration(void);
void cpstatement_part(void);
void cpstatement(void);
void cpassing(void);
```

```
void cpif(void);
void cpelse(void);
void cpwhile(void);
void cpcall(void);
void cpread(void);
void cpwrite(void);
void cpidentifier(void);
void cpexpresion(void);
void cpunary(void);
void cpterm(void);
void cpnumber(void);
void cpoperator(void);
void cprelation(void);
void cprelational_operator(void);
/*****************************************************************/

/******************DECLARACION DE DATOS*************************/
typedef struct listacprimero conjp;
struct listacprimero{
tokens cprimero;
char donde[40];
conjp *sig;
};
conjp *I=NULL;//puntero a la lista de conjuntos primero
int terminal=0;
/*****************************************************************/

conjp* nuevo(void)
{
   conjp *temp;
   temp=(conjp *)malloc(sizeof(conjp));
   if(!(temp))
     {
        printf("\n\tERROR: Insuficiente espacio de memoria");
        printf("\n\tSaliendo del programa...");
        system("pause");
        exit(-1);
     }
   return temp;
}
void cpliberar(conjp **list)
{
   conjp *temp=*list,*aux=NULL;
     while(temp!=NULL)
     {
        aux=temp->sig;
        free(temp);
        temp=aux;
     }
   *list=temp;
```

```
}
void insertar(tokens t,char *d)
{
   conjp *temp=NULL,*aux;
   aux=I;
   temp=nuevo();
   temp->cprimero=t;
   strcpy(temp->donde,d);
   temp->sig=NULL;
   if(aux==NULL)
      I=temp;
   else
   {
   while((aux->sig)!=NULL)
            aux=aux->sig;
   aux->sig=temp;
   }
}

void ver(conjp **p)
{
   conjp  *aux=*p;
   while(aux!=NULL)
   {
      //printf("%s->",aux->donde);
      printf("%s\n",tokenName[aux->cprimero]);
      aux=aux->sig;
   }
}

void atrapar(tokens t,char *donde)
{
   insertar(t,donde);
   terminal=1;
}

void conjunto_primero(char *n)
{
   if(strcmpi(n,"program_declaration")==0)
      cpprogram_declaration();
   else if(strcmpi(n,"block")==0)
     cpblock();
   else if(strcmpi(n,"declaration_part")==0)
     cpdeclaration_part();
   else if(strcmpi(n,"constant_declaration")==0)
      cpconstant_declaration();
   else if(strcmpi(n,"type_declarer")==0)
      cptype_declarer();
   else if(strcmpi(n,"variable_declaration")==0)
      cpvariable_declaration();
```

```
   else if(strcmpi(n,"procedure_declaration")==0)
      cpproc_declaration();
   else if(strcmpi(n,"statement_part")==0)
      cpstatement_part();
   else if(strcmpi(n,"statement")==0)
      cpstatement();
   else if(strcmpi(n,"assign_statement")==0)
      cpassing();
   else if(strcmpi(n,"if_statement")==0)
      cpif();
   else if(strcmpi(n,"else_part")==0)
      cpelse();
   else if(strcmpi(n,"while_statement")==0)
      cpwhile();
   else if(strcmpi(n,"call_statement")==0)
      cpcall();
   else if(strcmpi(n,"read_statement")==0)
      cpread();
   else if(strcmpi(n,"write_statement")==0)
      cpwrite();
   else if(strcmpi(n,"expression")==0)
      cpexpresion();
   else if(strcmpi(n,"term")==0)
      cpterm();
   else if(strcmpi(n,"unary_op")==0)
      cpunary();
   else if(strcmpi(n,"operator")==0)
      cpoperator();
   else if(strcmpi(n,"relation")==0)
      cprelation();
   else if(strcmpi(n,"relational_operator")==0)
      cprelational_operator();
   else
   printf("Termino Desconocido...");
   ver(&I);
   cpliberar(&I);
}

void cpprogram_declaration(void)
{
   cpblock();
   if(terminal==0)
   atrapar(PERIOD_,"Program_declaration");
}

void cpblock(void)
{
   atrapar(BEGIN_,"Block");
   if(terminal==0)
   {
```

```
    cpdeclaration_part();
    cpstatement_part();
    atrapar(END_,"");
    }
}

void cpdeclaration_part(void)
{
    //printf("\tDeclaration_part-> ");
    cpconstant_declaration();
}

void cpconstant_declaration(void)
{
   // printf("\tConstant_declaration-> ");
    cptype_declarer();
}
void cptype_declarer(void)
{
    atrapar(INTEGER_,"Type_declarer");
}

void cpvariable_declaration(void)
{
    atrapar(VAR_,"Variable_declaration");
    cptype_declarer();
}

void cpproc_declaration(void)
{
    atrapar(PROCEDURE_,"Procedure_declaration");
}

void cpstatement_part(void)
{
    //printf("\tStatement_part-> ");
    cpstatement();
}

void cpstatement(void)
{
    //printf("\tStatement-> ");
    cpassing();
    cpif();
    cpwhile();
    cpcall();
    cpread();
    cpwrite();
}
```

```
void cpassing(void)
{
   //printf("\tAssign_statement-> ");
   cpidentifier();
}
void cpif(void)
{
   atrapar(IF_,"If_statement");
}
void cpwhile(void)
{
   atrapar(WHILE_,"While_statement");
}
void cpcall(void)
{
   atrapar(CALL_,"Call_statement");
}
void cpread(void)
{
   atrapar(READ_,"Read_statement");
}
void cpwrite(void)
{
   atrapar(WRITE_,"Write_statement");
}
void cpidentifier(void)
{
   atrapar(ID_,"Identifier");
}

void cpelse(void)
{
   atrapar(ELSE_,"Else_part");
}
void cpexpresion(void)
{
   cpunary();
}
void cpunary(void)
{
   atrapar(NEGATE_,"Unary_op");
   atrapar(ABS_,"Unary_op");
}
void cpterm(void)
{
   atrapar(OPEN_,"Term");
   cpidentifier();
   cpnumber();
}
void cpnumber(void)
```

```
{
   atrapar(NUMBER_,"Number");
}
void cpoperator(void)
{
   atrapar(PLUSOP_,"Operator");
   atrapar(MINUSOP_,"Operator");
   atrapar(TIMESOP_,"Operator");
   atrapar(OVEROP_,"Operator");
   atrapar(MODOP_,"Operator");
}
void cprelation(void)
{
   atrapar(OPEN_,"Relation");
}
void cprelational_operator(void)
{
   atrapar(EQUAL_,"Relational_Operator");
   atrapar(NOTEQUAL_,"Relational_Operator");
   atrapar(LESS_,"Relational_Operator");
   atrapar(LESS_EQ_,"Relational_Operator");
   atrapar(GREAT_, "Relational_Operator");
   atrapar(GREAT_EQ_, "Relational_Operator");
}
```

Main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "Scan.h"
#include "Parser.h"
#include "conjPrimero.h"

int main(int argc, char *argv[])
{
  char nombre_archivo[20];
  char buscar_cj[20];
  char caracter_actual;
  int total_caract=0,cuenta_espacios=0,opcion=0;
  tokens tom;//Toma el valor devuelto por la funcion Scanner

  system("cls");
  system("title Conjunto Primero");
  system("color 9B");

  printf("\n\tNombre del archivo o fichero: ");
  fflush(stdin);
  gets(nombre_archivo);
  if((ptrtofile=fopen(nombre_archivo,"r"))==NULL)
  {
```

```
      printf("\n\tEl archivo: %s no se pudo abrir",nombre_archivo);
      printf("\n\tSaliendo del programa...\n\t");
      system("pause");
      exit(0);
}
else
  {
      do{
        printf("\n\t\t* * * *   MENU   * * * *\n");
        printf("1- Lista de Tokens y simbolos desconocidos.\n");
        printf("2- Resumen General de un input de entrada.\n");
        printf("3- Errores de Parseo.\n");
        printf("4- Definir Conjunto Primero de SLANG.\n");
        printf("\n\n\aOpcion: ");
        scanf("%d",&opcion);
       }while(opcion<1 && opcion >4);
      if(opcion==1)
      {
      system("cls");
      printf("\n\t * * * *   ERRORES ENCONTRADOS   * * * \n");
      while(!ferror(ptrtofile) && !feof(ptrtofile))
      {
         tom=scanner();
         insertar_tokens_en_lista(&cab,tom);
          if(flagEOF==1)
          {
             tom=scanner();
             insertar_tokens_en_lista(&cab,tom);
             flagEOF=0;
          }
      }
      if(flagerror==0)
         printf("\n\tNo se encontraron errores durante el SCANEO");
      printf("\n\n\t  * * * * *    SCANEO    * * * * * *\n");
      printf("\n\t  ---------------------------------------");
      visualizar(cab);
      printf("\n\t  ---------------------------------------");
      printf("\n\tFin del Scaneo...");
      liberar(&cab);
      fclose(ptrtofile);
    }//fin del if opcion==1

  }

if(opcion==3)
{
system("cls");
rewind(ptrtofile);
while(!ferror(ptrtofile) && !feof(ptrtofile))
      {
```

```
        tom=scanner();
        insertar_tokens_en_lista(&cab,tom);
          if(flagEOF==1)
          {
             tom=scanner();
             insertar_tokens_en_lista(&cab,tom);
             flagEOF=0;
          }
     }
printf("\n\n\t  * * * * * * *   PARSEO   * * * * * * *\n");
printf("\n\t  ---------------------------------------");
printf("\n\t  **********ERRORES ENCONTRADOS**********");
program_declaration();
  if(err==0)
    printf("\n\tNo se encontraron errores durante el PARSEO");
printf("\n\t  ---------------------------------------");
printf("\n\tFin del Parseo...");
liberar(&cab);
fclose(ptrtofile);
}

if(opcion==2)
{
kase2=1;
system("cls");
rewind(ptrtofile);
while(!ferror(ptrtofile) && !feof(ptrtofile))
     {
        tom=scanner();
        insertar_tokens_en_lista(&cab,tom);
          if(flagEOF==1)
          {
             tom=scanner();
             insertar_tokens_en_lista(&cab,tom);
             flagEOF=0;
          }
     }
program_declaration();
  if(err==0)
    printf("\n\tNingun ERROR fue encontrado durante el PARSEO");
rewind(ptrtofile);
while(!ferror(ptrtofile) && !feof(ptrtofile))
{
   caracter_actual=fgetc(ptrtofile);
    if(!(isgraph(caracter_actual)))//Cantidad de espacio en blancos
    {
      if(caracter_actual==' ')
             cuenta_espacios++;
      if(caracter_actual!=' ' && caracter_actual!='\n' && isspace(caracter_actual))
             cuenta_espacios++;
```

```
      }
     total_caract++;
   }
  printf("\n\n\t\t  * * * * * * * * * *   RESUMEN   * * * * * * * * * * *\n");
  printf("\n\t\t  ----------------------------");
  printf("\n\t\t1-Total de LINEAS: %d",numlin);
  printf("\n\t\t2-Total de TOKENS: %d",total_token-1);
  printf("\n\t\t3-Total de PALABRAS RESERVADAS: %d",total_pal_reser);
  printf("\n\t\t4-Total de SIMBOLOS ESPECIALES: %d",total_sim_espe);
  printf("\n\t\t5-Total de ERRORES:");
  printf("\n\t\t  *-De Scanneo: %d",total_err_scan);
  printf("\n\t\t  *-De Parseo: %d",total_err_parser);
  printf("\n\t\t6-Total de ESPACIOS EN BLANCO: %d",cuenta_espacios);
  printf("\n\t\t7-Total de CARACTERES: %d",total_caract-1);
  printf("\n\t\t8-Total de IDENTIFICADORES: %d",total_id);
  liberar(&cab);
  fclose(ptrtofile);
  }

  if(opcion==4)
  {
  system("cls");
  printf("\n\n\t");
  printf("Introduzca la produccion: ");
  fflush(stdin);
  gets(buscar_cj);
  printf("\n\tSu Conjunto Primero es:\n");
  conjunto_primero(buscar_cj);

  }
  printf("\n\n");
  system("PAUSE");
  return 0;
}
```