Tecnológico Nacional de México

Instituto Tecnológico de Veracruz

Ingeniería en Sistemas Computacionales

Fase Sintáctica del Compilador:

LIGHT

Como parte de la asignatura de:

Lenguajes y Autómatas I

Alumnos:

Rodríguez González Luis Uriel

Couary Juárez David Samuel

Catedrático:

M.E. Ofelia Gutiérrez Giraldi

Grupo:

6J1- "C". 14:00-15:00

H. Veracruz, Ver. 16 de mayo del 2017

Índice

Índice de Imágenes

Índice de Tablas

Introducción

Una de las grandes cosas que ha logrado el hombre en las últimas décadas, es el poder establecer una comunicación con las máquinas, y poder proporcionarles las instrucciones precisas de las tareas que quiere realizar, de las operaciones que desea ejecutar o de la información que desea desplegar. Sin embargo, como es bien sabido, los humanos hablamos un idioma y las computadoras otro, por lo que es indispensable contar con una especie de “traductor” que funja como intermediario en este proceso de comunicación.

Esta comunicación que existe entre el hombre y la máquina depende ampliamente de los lenguajes de programación y de los compiladores, pues todos los sistemas de software modernos que se ejecutan en nuestras computadoras fueron escritos en algún lenguaje de programación. Un lenguaje de programación, contiene sentencias que son más fáciles de comprender por las personas (llamadas programadores), y que son traducidas a sentencias que nuestra maquina pueda entender y ejecutar.

Es por eso que para que los programas escritos en cualquier lenguaje de programación sean ejecutados, estos deben estar traducidos en un “idioma” que nuestra maquina entienda. Los programas de software que realizan dicha traducción son esencialmente los *compiladores*.

A través de la historia de la computación, han existido, existen y existirán un gran número de lenguajes de programación (C, C++, C#, Pascal, Java, Cobol, Python, Ruby, etc.). Todos y cada uno de ellos poseen ciertas reglas, propósitos y sintaxis diferente, sin embargo, convergen en la idea ser lenguajes que nos ayuden a indicarle a las computadoras lo que queremos hacer, y que para ello necesitan ser compilados, es decir, traducidos.

Ha llegado el momento de crear nuestro propio lenguaje de programación, ya que, uno de los objetivos de la asignatura de Lenguajes y Autómatas I, es el diseño, creación e implementación de un compilador, en sus tres primeras fases: léxica,

sintáctica y semántica, y por ende ello implica la creación del lenguaje perteneciente a dicho compilador.

Para la implementación de un compilador, se aplican diversos fundamentos teóricos de esta asignatura tales como la teoría de autómatas, las expresiones regulares, las gramáticas, entre otros. Es por eso que para poder implementar nuestro compilador es indispensable que contemos con estos conocimientos previos.

En el presente trabajo hacemos hincapié en las dos primeras fases de un compilador: la fase léxica y la fase sintáctica. Como bien sabemos, en la primer fase se realiza una especia de “escaneo” del código fuente y hace la separación en estructuras individuales denominadas “tokens”. Por otro lado, en la segunda parte esos “tokens” se acomodan para formas estructuras gramaticalmente correctas.

Como ejemplos de “tokens” podemos mencionar las palabras reservadas, los delimitadores o los tipos de datos. Nosotros en la primera fase realizamos la definición de los “tokens” que pertenecerán a nuestro lenguaje, y es el analizador sintáctico implementado en JavaCC el que se encargará de realizar este proceso.

En la segunda fase, tras una minuciosa revisión de todas las posibilidades de operaciones e instrucciones que podría realizar el lenguaje, realizamos la gramática libre de contexto de light, misma que nos sirvió como base para terminar la segunda fase del compilador, y darle más sentido a los tokens encontrados en la primera fase.

A continuación, mostramos los resultados de la implementación de la implementación de la primera y segunda fase de nuestro compilador y de nuestro lenguaje llamado Light.

Marco Teórico

Traductores

Un traductor se define como un programa que traduce o convierte desde un texto o programa escrito en un lenguaje fuente hasta un texto o programa equivalente escrito en un lenguaje destino produciendo, si cabe, mensajes de error. Los traductores engloban tanto a los compiladores (en los que el lenguaje destino suele ser código máquina) como a los intérpretes (en los que el lenguaje destino está constituido por las acciones atómicas que puede ejecutar el intérprete).

Desde los orígenes de la computación, ha existido un abismo entre la forma en que las personas expresan sus necesidades y la forma en que un ordenador es capaz de interpretar instrucciones. Los traductores han intentado salvar este abismo para facilitarles el trabajo a los humanos, lo que ha llevado a aplicar la teoría de autómatas a diferentes campos y áreas concretas de la informática, dando lugar a los distintos tipos de traductores, entre los cuales se encuentran los traductores de idiomas, los intérpretes y los compiladores.

Compilador

Los compiladores son programas de computadora que traducen un lenguaje a otro. Un compilador toma como su entrada un programa escrito en su lenguaje fuente y produce un programa equivalente escrito en su lenguaje objetivo. Por lo regular, el lenguaje fuente es un lenguaje de alto nivel, tal como C o C++, mientras que el lenguaje objetivo es código objeto (también llamado en ocasiones código de máquina) para la máquina objetivo, es decir, código escrito en las instrucciones de máquina correspondientes a la computadora en la cual se ejecutará.

Un compilador es un programa muy complejo con un número de líneas de código que puede variar dependiendo el diseño y las funciones de este. Escribir un programa de esta naturaleza, o incluso comprenderlo, no es una tarea fácil, y la mayoría de los científicos y profesionales de la computación nunca escribirán un

compilador completo. No obstante, los compiladores se utilizan en casi todas las formas de la computación, y cualquiera que esté involucrado profesionalmente con las computadoras debería conocer la organización y el funcionamiento básicos de un compilador. Además, una tarea frecuente en las aplicaciones de las computadoras es el desarrollo de programas de interfaces e intérpretes de comandos, que son más pequeños que los compiladores pero utilizan las mismas técnicas. Por lo tanto, el conocimiento de estas técnicas tiene una aplicación práctica importante.

Fases de un compilador

Hasta ahora, se ha tratado al compilador como una caja simple que mapea un programa fuente a un programa destino con equivalencia semántica. Si abrimos esta caja un poco, podremos ver que hay dos procesos en esta asignación: análisis y síntesis.

La parte del análisis divide el programa fuente en componentes e impone una estructura gramatical sobre ellas. Después utiliza esta estructura para crear una representación intermedia del programa fuente. Si la parte del análisis detecta que el programa fuente está mal formado en cuanto a la sintaxis, o que no tiene una semántica consistente, entonces debe proporcionar mensajes informativos para que el usuario pueda corregirlo. La parte del análisis también recolecta información sobre el programa fuente y la almacena en una estructura de datos llamada tabla de símbolos, la cual se pasa junto con la representación intermedia a la parte de la síntesis.

La parte de la síntesis construye el programa destino deseado a partir de la representación intermedia y de la información en la tabla de símbolos.

Si examinamos el proceso de compilación con más detalle, podremos ver que opera como una secuencia de fases, cada una de las cuales transforma una representación del programa fuente en otro. En la figura 1 se muestra una descomposición típica de un compilador en fases. En la práctica varias fases

pueden agruparse, y las representaciones intermedias entre las fases agrupadas no necesitan construirse de manera explícita. La tabla de símbolos, que almacena información sobre todo el programa fuente, se utiliza en todas las fases del compilador.

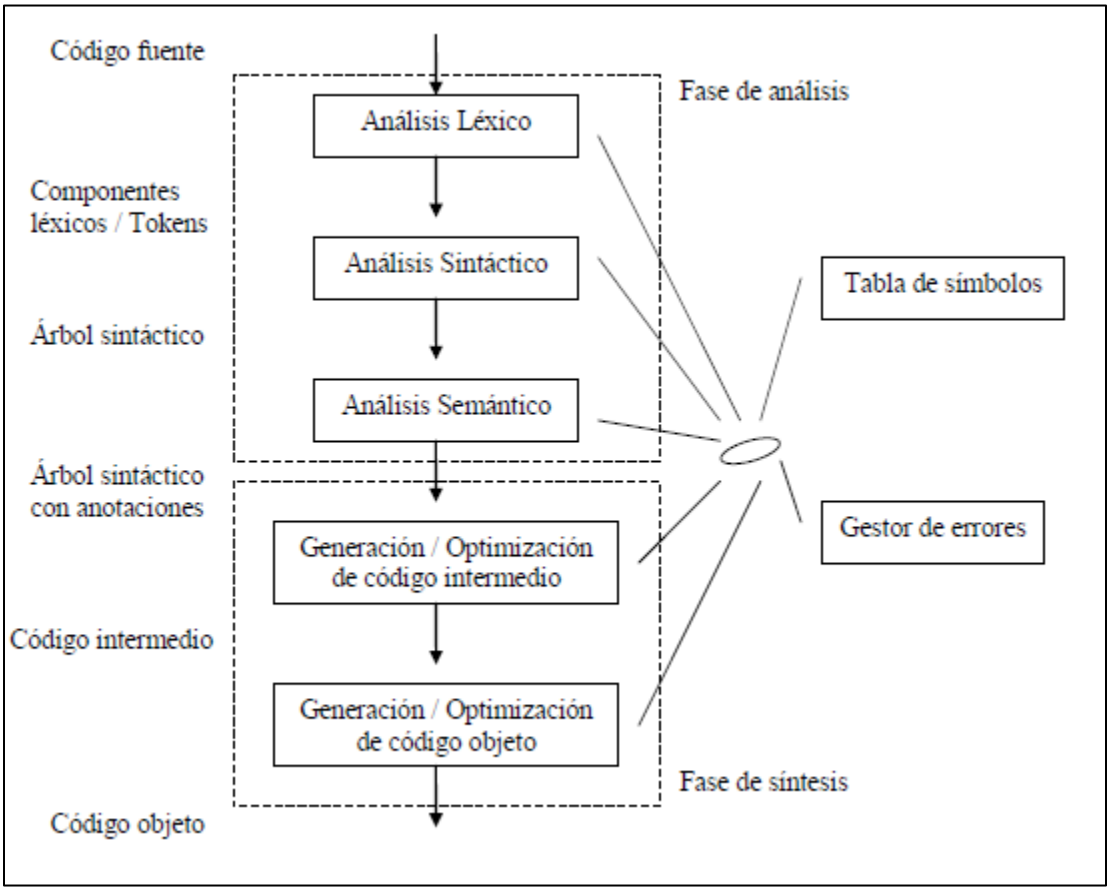

Imagen 1. Fases de un Compilador

Como ya se mencionó con anticipación, en el presenta trabajo nos enfocamos en la primera fase del compilador, que resulta ser la fase léxica, a continuación, la describiremos más a detalle.

Análisis Léxico

Esta fase del compilador efectúa la lectura real del programa fuente, el cual generalmente está en la forma de un flujo de caracteres. El rastreador realiza lo que se conoce como análisis léxico: recolecta secuencias de caracteres en unidades significativas denominadas tokens, las cuales son como las palabras de un lenguaje

natural, como el inglés. De este modo, se puede imaginar que un rastreador realiza una función similar al deletreo

El analizador de léxico lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias significativas, conocidas como lexemas. Para cada lexema, el analizador léxico produce como salida un token de la forma:

<nombre-token, valor-atributo>

que pasa a la fase siguiente, el análisis de la sintaxis. En el token, el primer componente nombre-token es un símbolo abstracto que se utiliza durante el análisis sintáctico, y el segundo componente valor-atributo apunta a una entrada en la tabla de símbolos para este token. La información de la entrada en la tabla de símbolos se necesita para el análisis semántico y la generación de código.

Para entender mejor como se realiza un análisis léxico, consideremos el siguiente ejemplo:

Considera la siguiente línea de código, que podría ser parte de un programa en C:

a [index] = 4 + 2

Este código contiene 12 caracteres diferentes de un espacio en blanco, pero sólo 8 tokens:

Valor	Tipo
a	Identificador
[	Corchete izquierdo
index	identificador
]	Corchete derecho
=	asignación
4	Número
+	Signo más
2	número

Tabla 1. Tabla valor-tipo de ejemplo

Cada token se compone de uno o más caracteres que se reúnen en una unidad antes de que ocurra un procesamiento adicional. Un analizador léxico puede realizar otras funciones junto con la de reconocimiento de tokens. Por ejemplo, puede introducir identificadores en la tabla de símbolos, y puede introducir literales en la tabla de literales (las literales incluyen constantes numéricas tales como 3.1415926535 y cadenas de texto entrecomilladas como "hola, mundo!").

Análisis Sintáctico

El analizador léxico tiene como entrada el código fuente en forma de una sucesión de caracteres. El analizador sintáctico tiene como entrada los lexemas que le suministra el analizador léxico y su función es comprobar que están ordenados de forma correcta (dependiendo del lenguaje que queramos procesar). Los dos analizadores suelen trabajar unidos e incluso el léxico suele ser una subrutina del sintáctico.

Al analizador sintáctico se le suele llamar párser. El párser genera de manera teórica un árbol sintáctico. Este árbol se puede ver como una estructura jerárquica que para su construcción utiliza reglas recursivas. La estructuración de este árbol hace posible diferenciar entre aplicar unos operadores antes de otros en la evaluación de expresiones. Es decir, si tenemos esta expresión en Java:

```
x = x * y – 2;
```

El valor de x dependerá de si aplicamos antes el operador producto que el operador suma. Una manera adecuada de saber qué operador aplicamos antes es elegir qué árbol sintáctico generar de los dos posibles.

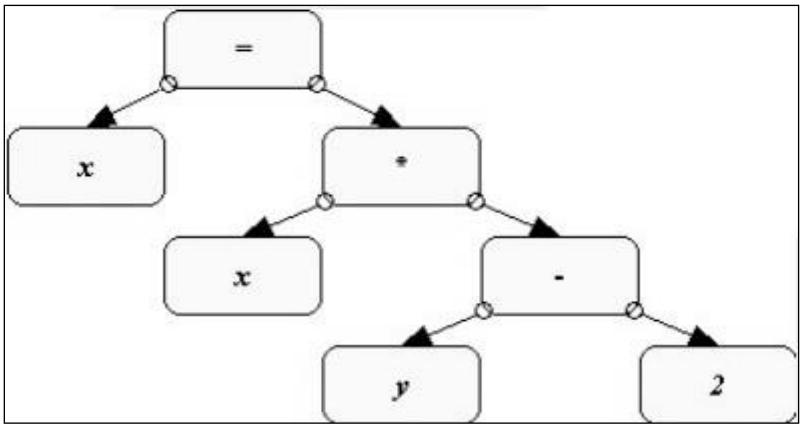

Imagen 2. Árbol Sintáctico 1

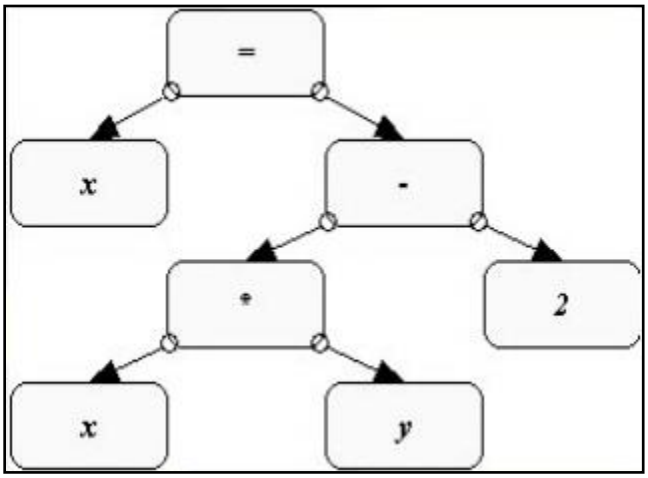

Imagen 3. Árbol Sintáctico 2

En resumen, la tarea del analizador sintáctico es procesar los lexemas que le suministra el analizador léxico, comprobar que están bien ordenados, y si no lo están, generar los informes de error correspondientes. Si la ordenación es correcta, se generará un árbol sintáctico teórico.

JavaCC

JavaCC (Java Compiler Compiler - Metacompilador en Java) es el principal metacompilador en JavaCC, tanto por sus posibilidades, como por su ámbito de difusión. Se trata de una herramienta que facilita la construcción de analizadores léxicos y sintácticos por el método de las funciones recursivas, aunque permite una notación relajada.

JavaCC integra en una misma herramienta al analizador lexicográfico y al sintáctico, y el código que genera es independiente de cualquier biblioteca externa, lo que le confiere una interesante propiedad de independencia respecto al entorno. Algunas de sus características son:

• Genera analizadores descendentes, permitiendo el uso de gramáticas de propósito general y la utilización de atributos tanto sintetizados como heredados durante la construcción del árbol sintáctico.

• Las especificaciones léxicas y sintácticas se ubican en un solo archivo. De esta manera la gramática puede ser leída y mantenida más fácilmente. No obstante, cuando se introducen acciones semánticas, recomendamos el uso de ciertos comentarios para mejorar la legibilidad.

• Admite el uso de estados léxicos y la capacidad de agregar acciones léxicas incluyendo un bloque de código Java tras el identificador de un token.

Lenguaje de programación Light

Nombre del lenguaje

Light. Nada más simple que eso. Light es una palabra inglesa que tiene dos principales traducciones al español: *ligero* o *luz.* Este término también puede usarse como algo que es liviano, tenue, claro, iluminado. Y eso es lo que pretende ser nuestro lenguaje de programación: un lenguaje sencillo, ligero, claro, fácil de entender, además hacemos alusión a la luz con un foco, lo cual se explica más adelante.

Logotipo

Imagen 4. Logotipo de Light

El logotipo de Light es representado a través de un bombillo o foco. Y usted se preguntará: - ¿esto qué tiene que ver con light? Pues dado que otra de las traducciones de light es luz, es por ello que decidimos usar un foco para representar esta palabra. Además, este hace referencia a la expresión: - ¡se me prendió el foco!, la cual coloquialmente se refiere a concebir una idea. Y pues como nosotros sabemos la programación está estrechamente relacionada con las ideas. ¿ahora encuentra usted la relación?

Historia

El lenguaje de programación Light fue creado por Luis Uriel Rodríguez González, Osvaldo Barradas Galeote y David Samuel Couary Juárez, alumnos de la carrera de Ingeniería en Sistemas Computacionales en el año 2017, durante su curso en la materia de lenguajes y autómatas I. Al principio se había decidido llamar al lenguaje RBC (acrónimos de las iniciales de sus apellidos), pero al encontrarlo poco creativo y nada representativo, se entró en el conflicto de encontrar un nombre adecuado para el lenguaje, que, aunque ya estaba en proceso de diseño, aun no tenía nombre. Sin embargo, en una de tantas sesiones se propuso el nombre de light y su significado y fue así como se decidió darle este nuevo nombre.

Sus creadores consideraron hacer un lenguaje de programación en español, con la intención que una persona que no sepa nada de inglés y que apenas esté empezando a relacionarse con el mundo de la programación pueda aprender de manera más rápida lo que es un lenguaje de programación y a cómo utilizarlo, basados en sus experiencias a la hora de utilizar por primera vez un lenguaje de programación.

Es por ello que las palabras reservadas de light de todas las instrucciones que maneja son completamente en español, siguiendo incluso una estructura muy parecida a un diagrama de flujo.

Propósito General

Crear un lenguaje de programación sencillo y fácil de usar, que utilice las funciones básicas de un lenguaje de estructurado.

Propósitos Específicos

- ✓ Utilizar palabras en español como palabras reservadas para facilitar la compresión del lenguaje.
- ✓ Facilitar la introducción a un lenguaje de programación a personas que nunca han trabajado con uno.
- ✓ Utilizar una estructura sencilla y clara, facilitando a los usuarios la codificación de sus diagramas de flujo a programas en este lenguaje.
- ✓ Utilizarse como un lenguaje sencillo de usar, que facilite la implementación de diversos tipos de algoritmos.
- ✓ Ser un lenguaje que emplee las mismas sentencias que algunos lenguajes como C o Java, para facilitar relacionarlo con otros lenguajes de programación.
- ✓ Crear un compilador que cuente con una interfaz sencilla, intuitiva y fácil de usar.

Tabla de símbolos

Palabras Reservadas Condicionales		
Valor	**Tipo**	**Descripción**
si	Palabra reservada si	Condicional ‘if’. Usado en toma de decisiones lógicas
sino	Palabra reservada sino	Condicional ‘else’. Usado para hacer contraste en las decisiones de ‘if’
elegir	Palabra reservada elegir	Condicional ‘switch’. Toma de decisiones múltiple.
caso	Palabra reservada caso	Instrucción ‘case’. Para expresar cada una de las opciones de una toma de decisiones múltiple.
salir	Palabra reservada salir	Instrucción ‘break’. Para salir de una toma de decisiones múltiple.
falta	Palabra reservada falta	Instrucción por ‘default’, para todo aquello que no encaje en los ‘casos’ de ‘elegir’

Tabla 2. Condicionales

Palabras Reservadas para tipos de datos		
Valor	**Tipo**	**Descripción**
ent	Palabra reservada ent	Declaración de tipo de dato entero
float	Palabra reservada float	Declaración de tipo de dato flotante
car	Palabra reservada car	Declaración de tipo de dato carácter
cad	Palabra reservada cad	Declaración de tipo de dato cadena
bool	Palabra reservada bool	Declaración de tipo de dato booleano

Tabla 3. Declaración de tipos de datos.

Palabras Reservadas E/S		
Valor	**Tipo**	**Descripción**
imprimir	Palabra reservada imprimir	Acción imprimir
leer	Palabra reservada leer	Acción leer entrada
escribir	Palabra reservada escribir	Acción escribir salida

Tabla 4. Entrada-Salida

Palabras Reservadas Ciclos		
Valor	**Tipo**	**Descripción**
mientras	Palabra reservada mientras	Ciclo 'while'. Para realizar ciclos opcionales.
hasta	Palabra reservada hasta	Ciclo 'for'. Para realizar ciclos un numero conocido de veces.
hacer	Palabra reservada hacer	Ciclo 'do'. Para realizar un ciclo por lo menos una vez.

Tabla 5. Ciclos

Palabras Reservadas de Estructura		
Valor	**Tipo**	**Descripción**
inicio	Palabra reservada inicio	Indica el inicio del programa
fin	Palabra reservada fin	Indica la finalización del programa

Tabla 6. Estructura

Delimitadores		
Valor	**Tipo**	**Descripción**
(	Delimitador (	Corchete izquierdo
)	Delimitador)	Corchete derecho
{	Delimitador {	Llave izquierda
}	Delimitador }	Llave derecha

‘	Delimitador ‘	Para encerrar caracteres
“	Delimitador “	Para delimitar cadenas
@	Delimitador @	Para comentarios
;	Delimitador ;	Para finalizar una sentencia
.	Delimitador .	Punto

Tabla 7. Delimitadores

Operadores Aritméticos		
Valor	**Tipo**	**Descripción**
+	Operador aritmético suma +	Sumar
-	Operador aritmético resta -	Restar
*	Operador aritmético producto *	Multiplicar
/	Operador aritmético cociente /	Dividir
%	Operador aritmético residuo %	Residuo de una división
^	Operador aritmético potencia ^	Potencia
=	Operador aritmético asignación =	Asignación
++	Operador aritmético aumento ++	Aumento en 1
--	Operador aritmético decremento --	Decremento en 1

Tabla 8. Operadores Aritméticos

Operadores Lógicos		
Valor	**Tipo**	**Descripción**
?	Operador Lógico NOT ?	Operación lógica NOT
&	Operador Lógico AND &	Operación lógica AND
||	Operador Lógico OR ||	Operación lógica OR

Tabla 9. Operadores Lógicos

Operadores de Comparación		
Valor	**Tipo**	**Descripción**
<	Operador comparativo menor <	Compara si es menor
>	Operador comparativo mayor >	Compara si es mayor
<=	Operador comparativo menor igual<=	Compara si es menor o iguales
>=	Operador comparativo mayor igual >=	Compara si es mayor o iguales
==	Operador comparativo igual ==	Compara si son iguales
?=	Operador comparativo diferente ?=	Compara si son diferentes

Tabla 10. Operadores de comparación

Tipos de datos		
Valor	**Tipo**	**Descripción**
Número enteros positivos y negativos	Tipo de dato entero	Números enteros
Números reales positivos y negativos	Tipo de dato flotante	Números decimales
Verdadero o Falso	Tipo de dato booleano	Datos booleanos
nulo	Tipo de dato nulo	Datos vacíos
Cadenas de caracteres encerradas entre comillas. Acepta acentos y la letra ñ.	Tipo de dato cadena	Cadenas
Un solo carácter encerrado entre comillas simples.	Tipo de dato caracter	caracteres

Tabla 11. Tipos de Dato

identificador		
Valor	**Tipo**	**Descripción**
Cadenas de caracteres que inicien con minúscula. Pueden incluir números o letras mayúsculas. Acepta acentos y la letra ñ.	identificador	Nombre de las variables

Tabla 12. Identificadores

Gramática del Lenguaje

La gramática de nuestro lenguaje se define y funciona de la siguiente manera:

Gramática (A...Z = variables, Palabras que inicien con mayúscula son variables, palabras en minúsculas son tokens, los símbolos como "; () {} ," son tokens)

S → B inicio {Instrucciones} fin B

B → comentario B | ε

Instrucciones → Declaraciones Instrucciones

|Asignaciones Instrucciones

|Ciclos Instrucciones

|Condicionales Instrucciones

|Es Instrucciones

|Comentarios Instrucciones

|ε

Declaraciones → Declara ident;

|Declara ident asignacion Operacion;

Declara → ent |flot |car |cad |bool

Dato → num |flotante |booleano |nulo |cadena |carácter |Neg

Neg → resta C

C → num | flotante | ident

Operacion → Operando D

D → Operador Operando D | ε

Operador → suma |resta |multiplicar |dividir |modulo |potencia

Operando → ident |Dato |(Operacion)

Asignaciones → ident E;

E → F | G

F → aumento | decremento

G → asignacion Operacion

Comentarios → comentario

Es → Printwrite | Scan

Printwrite → H (Concatenar);

H → escribir | imprimir

Concatenar → I J

I → Dato | ident

J → suma I J | ε

Scan → leer(Tipodato, ident);

Tipodato → ent |flot |car |cad |bool

Condicionales → Ifelse | Switchh

Ifelse → si(Condicion) {Instrucciones} K

K → sino{Instrucciones} | ε

Switchh → elegir(ident) {Casos Defaultt}

Casos → caso(Dato) {Instrucciones Breakk} Casos | ε

Breakk → salir; | ε

Defaultt → falta{Instrucciones} | ε

Condicion → Condicional L

L → Operadorlogico Condicional L | ε

Operadorlogico → no |o |y

Condicional → Comparacion

Comparacion → Operacion Comparador Operacion

Comparador → menor |mayor |menorigual |mayorigual |igual |diferente

Ciclos → Wwhile |Dowhile |Ffor

Wwhile → mientras(Condicion){Instrucciones}

Dowhile → hacer{Instrucciones} mientras(Condicion);

Ffor → hasta(M Condicion; Control) {Instrucciones}

M → Asignaciones | Declaraciones

Control → ident N

N → F | O | P

O → asignacion Operacion

P → Q asignacion R

Q -> suma | resta

R → num | Neg

Estructura General del lenguaje

Dado que uno de los objetivos específicos de este lenguaje de programación es facilitar la codificación de algoritmos, específicamente diagrama de flujo en un lenguaje de programación, la estructura general de un programa en lenguaje Light es el siguiente:

```
Inicio
{
        …
Bloque de instrucciones
        …
}
fin
```

Imagen 5. Estructura general de Light

Interfaz Gráfica del Compilador

Otro de los requerimientos importantes de este proyecto, es realizar la interfaz gráfica del compilador, por lo que se decidió crear un IDE sencillo pero funcional,

además que fuese fácil de usar y que tuviera las funciones básicas de un compilador.

Su diseño se basó en el funcionamiento de otros compiladores y editores de texto como Geany, Borland C++, Dev C++, Notepad++, Sublime Text entre otros.

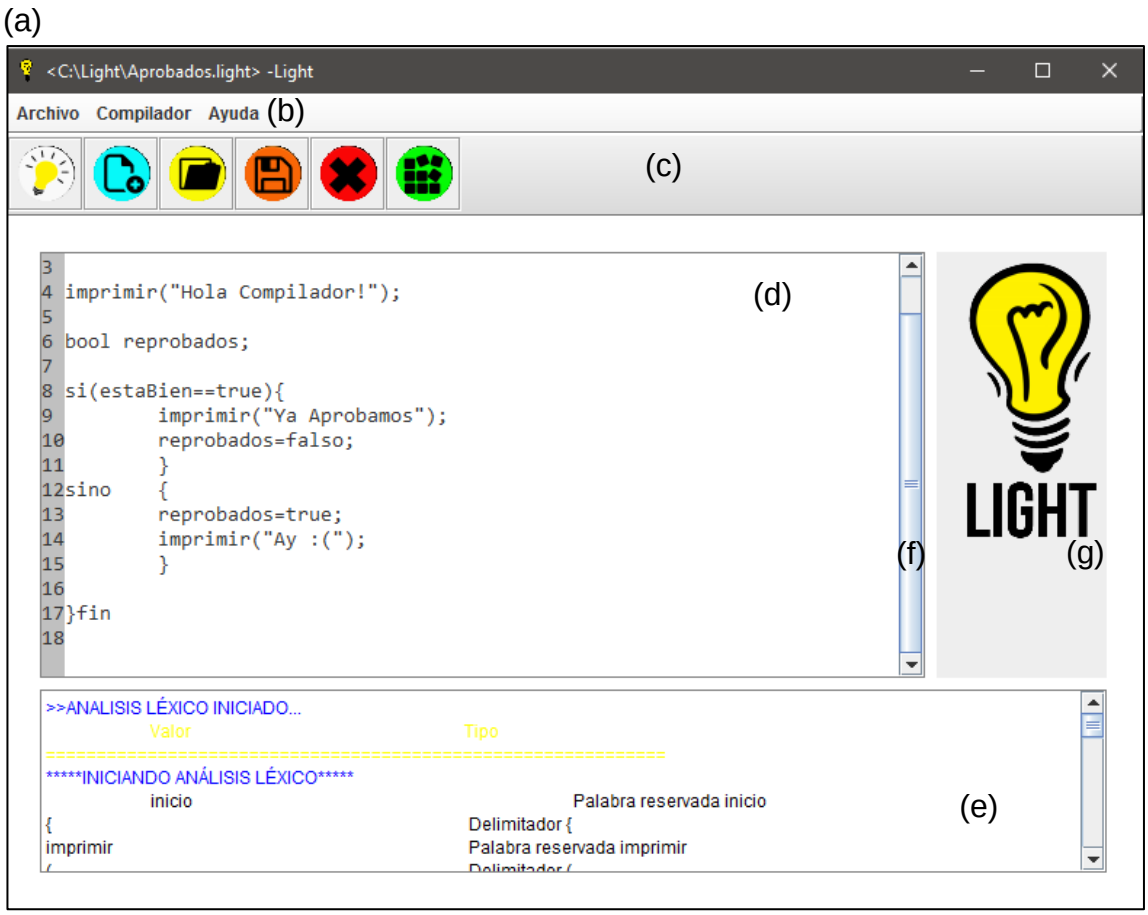

Imagen 6. Interfaz gráfica del compilador

Como podemos observar,el diseño se basa en la sencillez y ligereza, pero sin sacrificar la funcionalidad. A continuación, describimos los elementos más importantes y las funciones de dicho IDE.

Elementos

(a) Título. Nos muestra la ruta del archivo que estamos editando en este momento, así como el nombre del programa.

(b) Barra de Navegación. Usada para acceder a las diferentes opciones de menú.

(c) Barra de Herramientas. Contiene los botones para realizar las funciones del compilador. (Se explican más adelante).

(d) Panel de entrada. Dentro de este se escribe el código fuente del programa codificado en este lenguaje de programación.

(e) Panel de salida. Nos muestra los resultados del análisis del programa y en caso de ser necesario, los errores encontrados en el código fuente.

(f) Barras de Desplazamiento. Para desplazarse por los paneles de entrada y salida.

(g) Icono. El icono de nuestro lenguaje, únicamente con fines estéticos.

Barra de Herramientas

Imagen 7. Barra de Herramientas de Light

(1) Nuevo. Sirve para crear un nuevo código fuente. Nos ofrece la opción de salvar el trabajo que estemos realizando para evitar la pérdida de información.

(2) Abrir. Permite abrir un código fuente del lenguaje. Estos deben estar guardados en un archivo con extensión **.light.**

(3) Guardar. Permite guardar en una carpeta de fácil acceso los códigos fuente creados en la interfaz gráfica.

(4) Salir. Salir de la aplicación.

(5) Compilar. La función más importante del IDE, hasta el momento se encarga de realizar el análisis léxico y sintáctico del código fuente. En caso de haber errores los muestra en el panel de salida.

(6) Light. Una opción que hace honor al nombre del lenguaje. Es mejor que usted descubra por sí mismo su funcionalidad.

Conclusiones

A través del desarrollo de nuestro compilador, hemos aprendido la importancia de conocer las bases teóricas de la materia para poder aplicarlas en el diseño de nuestro lenguaje.

Por ejemplo, al saber cómo funciona la fase léxica, comprendemos la manera en que debe de ser programado nuestro compilador, y las funciones que debe realizar el analizado léxico.

Por otro lado, la fase sintáctica nos ayudó a comprender la manera en que los tokens se agrupan de acuerdo a una gramática para formar las estructuras de nuestro programa.

Un conocimiento que aplicamos fue el de las expresiones regulares, pues la definición de la estructura de algunos tokens, requerían de ellas, como por ejemplo las cadenas de caracteres o los números flotantes. Comprender la lógica de las expresiones regulares y aplicarlo en nuestro compilador fue muy interesante.

Aprendimos que construir un compilador no es una tarea sencilla, sin embargo, conocer su funcionamiento y sus etapas internas, nos ayudan a comprender mejor la manera en que leen, analizan, buscan errores, generan código, optimizan y traducen.

Hasta el momento ya hemos dado un paso más en la construcción de nuestro compilador, pues ya se han definido las dos primeras etapas de nuestro compilador, dejando todo listo para la siguiente fase.

Bibliografía

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2008). *Compiladores. Principios, Técinas y Herramientas*. México: Pearson Educación.

Gálvez Rojas, S., & Mora Mata, M. Á. (2005). *Java a Tope: Traductores y Compiladores con Lex/Yacc, JFlex/Cup y JavaCC.* Malaga, España: Universidad de Málaga.

Ruiz Catalán, J. (2010). Compiladores. Teoría e Implementación. Editorial Grupo Ramírez Cogollor.

Louden, K. (2004). *Construcción de Compiladores: Principios y Práctica.* Editorial Cengage Learning.