Trabajo práctico especial 1 y 2

Diseño de compiladores I

Lucas Marquez

lucasasecas@gmail.com

Grupo 36

# Introduccion

En el presente informe se describirán los componentes de un programa que funciona como

analizador lexico­sintactico así como también las decisiones de diseño que se tomaron para

implementarlo.

El programa aquí expuesto se puede dividir en dos partes. Por un lado se tiene un componente

analizador léxico que se encargara de ir leyendo caracteres de un archivo de texto y mediante

una el uso de una máquina de estados va devolviendo los tokens que va encontrando. El

segundo componente es un analizador sintáctico que hace uso del analizador léxico para ir

recogiendo tokens que este último le va dando y así armar las reglas gramaticales previamente

definidas.

La implementación del analizador sintáctico es prefabricada por la aplicación BYacc1

# Decisiones de diseño e implementación

Como se menciona en la sección anterior, el analizador está compuesto por un analizador

léxico y uno sintáctico.

En esta sección se muestra por separado las decisiones de diseño que se tomaron para cada

analizador (lexico y sintactico) asi como tambien la descripcion del diseño resultante.

Cabe destacar que ambas implementaciones se realizaron en el lenguaje de programación

JAVA por lo cual tiene un enfoque de programación orientada a objetos.

## Analizador Lexico:

El analizador léxico está compuesta por una clase principal llamada AnalizadorLexico la cual

implementa el método GetNextToken() que es el encargado de retornar un token cuando se

reconoce.

Las reglas léxicas están definidas en un autómata de estados implementado mediante una

matriz de transición de estados. Cada celda de dicha matriz almacena un componente de tipo

**Estado** encargada de almacenar el estado siguiente asi como tambien una lista de acciones

que se deben realizar antes de hacer la transición de estado.

Las acciones que se realizan en cada transición están implementadas por las clases que

heredan de la clase **Acción** donde cada una de ellas implementa el método ejecutar que realiza

acciones sobre el mismo analizador léxico.

Para el desarrollo también se utilizo unas clases auxiliares que sirven de apoyo a la

implementación. A continuación se detallan las más importantes.

### Tabla de símbolos

Esta clase implementa una tabla de símbolos que almacena instancias de la tabla

TuplaTablaSimbolos. En esta clase se implementan las funcionalidades encargadas de

lectura/escritura de dichas tuplas.

### TuplaTablaSimbolos:

Cumple la función de tupla de la tabla de símbolos y es un contenedor de los datos que

adicionales que se desean almacenar para un identificador. Tiene la particularidad de poseer un crecimiento horizontal dinamico.

### BufferedCarStream:

Esta clase implementa un buffer que tiene como entrada un input stream y ofrece

funcionalidades que son de utilidad para el analizador léxico como el método retract que da la

posibilidad de retroceder n caracteres hacia atrás de los que ya se leyeron.

## Analizador sintactico:

Para el diseño del analizador sintáctico no se tomaron muchas decisiones a nivel de análisis

detallado puesto que la herramienta BYacc genera el parser automáticamente.

Cabe destacar que se redefinió el método del parser yylex() y el constructor del parser donde

en el primero se utiliza una instancia de la clase AnalizadorLexico.

# Diagrama de transición de estados

El siguiente diagrama expresa el autómata con sus estados y arcos.

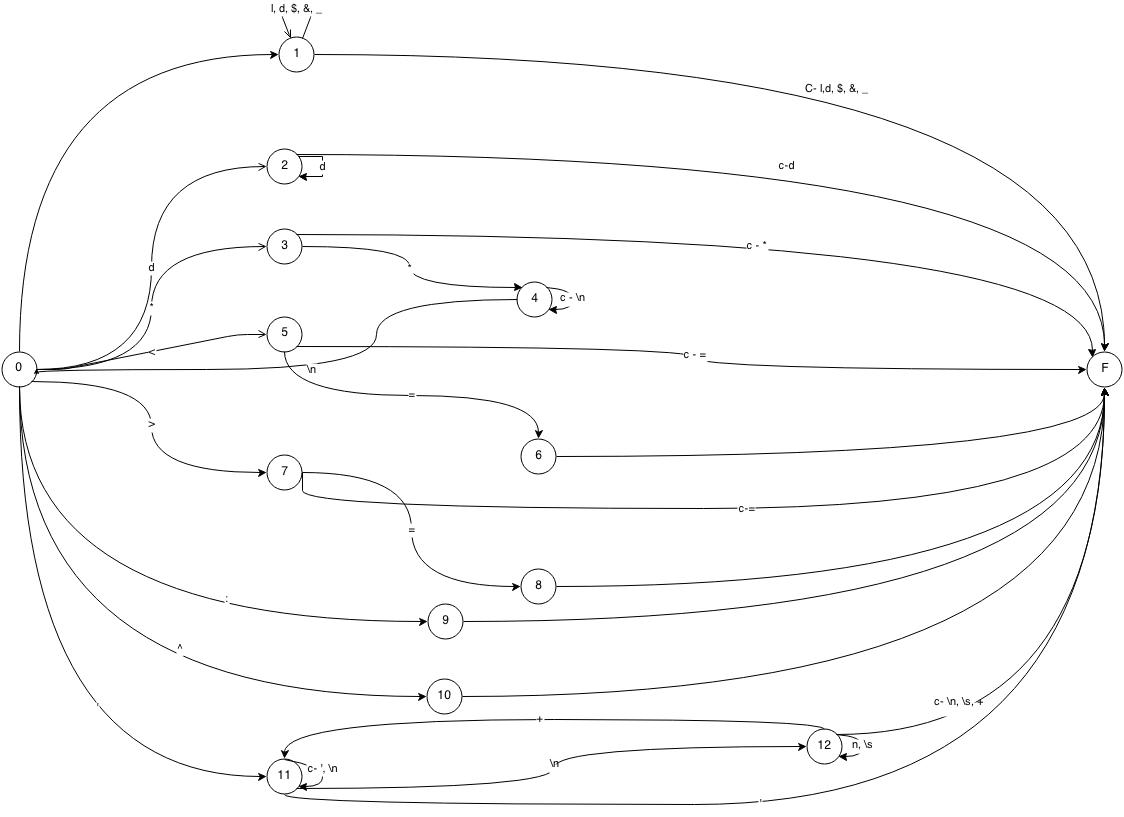
Por cuestiones de legibilidad no se expresó de forma completa el diagrama por lo cual a

continuación se pondrán algunas cuestiones que deberán tenerse en consideración.

* Cuando se lee un salto de línea, un tabulador o un espacio en blanco, se irá desde el

estado 0 al estado 0.

* El caracter # representa cualquier carácter que no esté soportado en el alfabeto.
* El fin de archivo se interpretara como un carácter más.



# Matriz de transición de estados

En la siguiente matriz se expone la matriz de transición de estados que representa al autómata

pero con otra estructura.

Las acciones semánticas asociadas a cada transición estas expresada como una conjunción

de varias acciones semánticas ya que varias de ellas se podían dividir en varias acciones con

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | l | d | + | - | \* | / | < | | > | | : | | = | | ^ | | ( | | ) | | { | | } | | , | | ; | | ' | | . | | /n | | \t,\s | | &, \_, $ | | | # | |
| 0 | 1 | 2 | F | F | 3 | F | | 5 | | 7 | | 9 | | F | | 10 | | F | | F | | F | | F | | F | | F | | 11 | | F | | 0 | | 0 | | F | 0 | |
| 1 | 1 | 1 | F | F | F | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | 1 | F | |
| 2 | F | 2 | F | F | F | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | F | |
| 3 | F | F | F | F | 4 | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | F | |
| 4 | 4 | 4 | 4 | 4 | 4 | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 0 | | 4 | | 4 | 4 | |
| 5 | F | F | F | F | F | F | | F | | F | | F | | 6 | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | F | |
| 6 | F | F | F | F | F | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | F | |
| 7 | F | F | F | F | F | F | | F | | F | | F | | 8 | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | F | |
| 8 | F | F | F | F | F | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | F | |
| 9 | F | F | F | F | F | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | F | |
| 10 | F | F | F | F | F | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | F | |
| 11 | 11 | 11 | 11 | 11 | 11 | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | F | | 11 | | 12 | | 11 | | 11 | 11 | |
| 12 | F | F | 11 | F | F | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | F | | 12 | | 12 | | F | F | |

una granularidad más fina.

# Acciones semánticas

En esta sección se listará y detalla de forma breve cada una de las 6 acciones semánticas

implementadas.

Como se mencionó en la sección anterior, por cada transición de estados se utilizaron una o

una combinación de acciones semánticas

Accion1: Se crea el nuevo token que se va a utilizar (inicializacion).

Accion2: Se agrega el carácter leído al token.

Accion3: Se verifica que el token leído tipo identificador no sobrepase el limite de largo permitido, en caso de hacerlo lo trunca

Accion4: se verifican los limites del token leído tipo constante.

Accion5: se devuelve un mensaje de error que fue previamente cargado.

Accion6: rollback: esta acción retirna en uno el lector de carácter.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | l | d | + | - | \* | / | < | > | : | = | ^ | ( | ) | { | } | , | ; | ' | /n | /t, /s | $ | # | EOF |
| 0 | Ac1 Ac2 | Ac1 Ac2 | Ac1 Ac2 | Ac1 Ac2 | Ac1 Ac2 | Ac1 Ac2 | Ac1 Ac2 | Ac1 Ac2 | Ac1 Ac2 | Ac1 Ac2 | Ac1 Ac2 | Ac1 Ac2 | Ac1 Ac2 | Ac1 Ac2 | Ac1 Ac2 | Ac1 Ac2 | Ac1 Ac2 | Ac1 |  |  | Ac1 Ac2 |  |  |
| 1 | Ac2 | Ac2 | Ac3 | Ac3 | Ac3 | Ac3 | Ac3 | Ac3 | Ac3 | Ac3 | Ac3 | Ac3 | Ac3 | Ac3 | Ac3 | Ac3 | Ac3 | Ac3 | Ac3 | Ac3 | Ac2 | Ac3 | Ac3 |
| 2 | Ac4 | Ac2 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 | Ac4 |
| 3 | Ac6 | Ac6 | Ac6 | Ac6 |  | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 |
| 4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac2 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 |
| 6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 7 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac2 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 | Ac6 |
| 8 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 9 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac2 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 |
| 10 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac2 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 | Ac5 |
| 11 | Ac2 | Ac2 | Ac2 | Ac2 | Ac2 | Ac2 | Ac2 | Ac2 | Ac2 | Ac2 | Ac2 | Ac2 | Ac2 | Ac2 | Ac2 | Ac2 | Ac2 | Ac8 |  | Ac2 | Ac2 | Ac2 | Ac5 |

# Descripción del desarrollo del analizador sintáctico

Esta sección se especificarán aspectos relacionados con la creación de la gramática así como

tambien los problemas encontrados a la hora de generar el parser a partir de dicha gramática.

El principal problema encontrado fueron los problemas shift/reduce que se fueron dando durante

la definición de la gramática. Si bien la mayoría se resuelve simplemente haciendo una

gramática mejor modularizada y siguiendo la pila de estados, también se tuvo que utilizar

recursos provistos por la herramienta yacc para que el comportamiento ante estos conflictos

sea el esperado.

A continuación se presenta el conflicto más clásico que se puede encontrar en internet y la

solución que se halló para dicho problema:

Se tiene la siguiente regla gramatical para la selección.

…

**Selección: SI ‘(’Comparacion’)’ ENTONCES SentenciasEntonces SINO SentenciasSino**

**| SI ‘(’Comparacion’)’ ENTONCES SentenciasEntonces**

…

La definición de la gramatica expuesta genera un conflicto shift/reduce cuando se genera

el parser.

La solución implementada fue la siguiente

1. En la sección de definiciones del documento donde se escribe la gramática

escribir las declaraciones

. . .

**%nonassoc IT**

**%nonassoc SINO**

. . .

2. En la declaración de la sentencia se procede de la siguiente manera

. . .

**Selección: SI ‘(’ Comparacion ‘)’ ENTONCES Sentencias SINO Sentencias**

**| SI ‘(’ Comparacion ‘)’ ENTONCES Sentencias %prec IT**

**;**

. . .

Con esto se esta logrando que la sentencia si else tenga una precedencia igual que IT

que a su vez se le dio una precedencia mayor que al token **SINO** (por ser definido primero). De

esta forma primero se debe reducir el else antes de seguir con el **SI** y asi se quita la

ambigüedad.

# Lista de elementos no terminales

En esta sección se mostrará el BNF que expresa las reglas gramaticales así como también la

descripción de los elementos no terminales.

Programa: Declaraciones Sentencias

| Declaraciones error

;

Declaraciones: Declaraciones Declaracion

| Declaracion

;

Declaracion: Declaracion\_reg

| Declaracion\_var

;

Declaracion\_reg: REGISTRO Cuerpo\_reg Lista\_var\_reg ';'

;

Lista\_var\_reg: Lista\_var\_reg ',' ID

| ID

;

Cuerpo\_reg: '{' Declaraciones\_reg '}'

;

Declaraciones\_reg: Declaraciones\_reg Declaracion\_unit

| Declaracion\_unit

;

Declaracion\_unit: Tipo ID ';'

;

Tipo: ENTERO

| ENTEROL

;

Lista\_var: Lista\_var ',' ID

| ID

;

Declaracion\_var: Tipo Lista\_var ';'

;

Sentencias: Sentencias Sentencia

| Sentencia

;

Sentencia: Asignacion

| Seleccion

| Iteracion

| Salida

;

Asignacion : AsignacionIzq ASIGN Expresion ';'

;

AsignacionIzq: ID

;

Expresion: Expresion '+' Termino;

| Expresion '-' Termino

| Termino

;

Termino: Termino '\*' Factor

| Termino '/' Factor

| Factor

;

Factor: ID

| CTE

| CampoReg

| CTENeg

;

CTENeg: '-' CTE

;

CampoReg: ID '.' ID

;

Seleccion: CabeceraSi BloqueEntonces SINO BloqueSino

| CabeceraSi BloqueEntonces %prec IT

;

CabeceraSi: SI '(' Comparacion ')' ENTONCES; {$$ = $1;}

;

Comparacion: Expresion COMP Expresion

;

BloqueEntonces: Sentencia | BloqueSentencias

;

BloqueSentencias: '{' Sentencias '}'

;

BloqueSino: BloqueSentencias

| Sentencia

;

Iteracion: CabeceraIteracion BloqueIteracion

;

CabeceraIteracion: MIENTRAS '(' Comparacion ')' ITERAR

;

BloqueIteracion: Sentencia

| BloqueSentencias

;

Salida: IMPRIMIR '(' STR ')' ';' {$$ = $1;}

;

* Programa: es el componente raiz el cual es definido por una lista de declaraciones

seguido de una lista de sentencias.

* Declaraciones: es una lista de una o más declaraciones.
* Declaración: es una declaración que se puede definir como una declaración de variables

una declaración de función.

* Declaracion\_Var: este no terminal es definido por un tipo seguido de una lista de

variables y debe terminar con el no terminal ‘;’.

* Tipo: es definido únicamente por el no terminal **entero** o **entero\_l**
* ListaVariables: la lista de variables(**ID**) esta definida por un unico no terminal ID o por

varios concatenadas. Las variables están separadas por el componente no terminal ‘,’.

* Declaracion\_reg: este no terminal se define como el no terminal **registro** seguido
* de {, y las declaraciones del registro }.
* Cuerpo\_reg: se defino como {, y las declaraciones del registro }.
* Declaraciones\_reg: Son las declaraciones internas del registro. Aca no se tiene lista de variables ya que no se especifico en la consigna
* Sentencias: es una lista de una o mas sentencias.
* Sentencia: es una sentencia que puede ser definida como asignación, Selección,

Iteración, Salida pantalla(imprimir ). Todas esas declaraciones excepto la de

selección e iteración termina con el terminal ‘;’

* Asignacion: una sentencia asignación se define como un identificador seguido del

símbolo terminal ‘:=’ y del lado derecho de éste una expresión.

* Seleccion: éste no terminal esta definido por el terminal SI seguido de una comparación

almacenada entre paréntesis, el terminal ENTONCES, un bloque de sentencias, el terminal

SINO (opcional) y otro bloque de sentencias en caso de que se encuentre el TOKEN

SINO.

* Iteracion: se define como una cabecera de iteración que es el token MIENTRAS seguido de una comparación encerrado por paréntesis. Seguido a la cabecera de iteración (Cabecera Iteración) le sigue la palabra recerbada ITERAR y luego un bloque de sentencias.

# Conclusion

Como conclusión final de esta primera parte se puede decir que la parte más complicada no

esta en la implementación sino en el diseño del mismo. Los errores más difíciles de controlar

están dados por reglas gramaticales mal diseñadas.

Saber el algoritmo que se utiliza para leer las reglas puede ser de gran ayuda entender el

porqué de ciertos problemas y conflictos que se generan.