Universidad Nacional del Centro de la
Provincia de Buenos Aires

FACULTAD DE CIENCIAS EXACTAS

Ingeniería de Sistemas

Trabajos Prácticos 1 y 2

Diseño de Compiladores I

Docente: Mailén González

GRUPO 17
Abraham, Maria Belen: abraham.belen.99@gmail.com
Piu, Florencia Jesus: fpiu@alumnos.exa.unicen.edu.ar
Silvestri Ayala, Giuliana Estefania: gsilvestriayala@alumnos.exa.unicen.edu.ar

7/10/24

ÍNDICE

Introducción

El siguiente proyecto tiene como objetivo desarrollar un compilador en Java utilizando YACC (Yet Another Compiler Compiler). Un compilador es una herramienta fundamental en el desarrollo de software, ya que permite la traducción de lenguajes de alto nivel, entendibles para los programadores, a lenguajes de bajo nivel, entendibles para el hardware.

En esta primera etapa el proyecto se enfoca en el desarrollo del Analizador Léxico, una de las primeras etapas de un compilador, que se encarga de dividir el código fuente en tokens reconocibles y procesables por las etapas posteriores del compilador.

Además, se ha implementado también un Analizador Sintáctico, cuya función es tomar los tokens generados por el analizador léxico y organizarlos en estructuras jerárquicas basadas en las reglas de la gramática del lenguaje fuente. Estas reglas definen la correcta organización de los tokens en expresiones y declaraciones válidas, asegurando que el código fuente respete las estructuras sintácticas establecidas.

Temas Particulares

1. **Enteros (16 bits)**: Constantes enteras con valores entre -2^{15} y -2^{15}-1. Se debe incorporar a la lista de palabras reservadas la palabra **integer**.

6. Punto flotante de 64 bits: Números reales con signo y parte exponencial. La parte exponencial puede estar ausente. Si está presente, el exponente comienza con la letra **"d"** (minúscula) y el signo del exponente es opcional. Su ausencia implica signo positivo para el exponente.
La parte entera, la parte decimal, y el '.' son obligatorios.

Ejemplos válidos: 1.0 0.6 -1.2 3.0d–5 2.0d+34 2.5d-1 13.0 0.0 1.2d10
Considerar el rango 2.2250738585072014d-308 < x < 1.7976931348623157d+308 ∪
-1.7976931348623157d+308 < x < -2.2250738585072014d-308 ∪ 0.0

8. Constantes hexadecimales, que se escribirán comenzando con 0x, y serán una secuencia de dígitos que pueden ir de 0 a 9 y letras de la A a la F. El rango para las constantes hexadecimales será el mismo que el del tipo entero asignado al grupo:

a. Para **integer:** -0x8000 a 0x7FFF

10. Cadenas multilínea: Cadenas de caracteres delimitadas por corchetes. Estas cadenas pueden ocupar más de una línea. (En la tabla de símbolos se guardará la cadena sin los saltos de línea).
Ejemplo: [¡Hola
Mundo!]

11. Incorporar a la lista de palabras reservadas la palabra **TYPEDEF**.

13. Incorporar a la lista de palabras reservadas la palabra **WHILE**.

18. Asignaciones múltiples: Se deben reconocer asignaciones múltiples, que permitan una lista de variables, separadas por comas (","), del lado izquierdo de la asignación, y una lista de expresiones, separadas por coma (",") del lado derecho.

21. Incorporar a la lista de palabras reservadas la palabra **PAIR**.

23. Incorporar a la lista de palabras reservadas la palabra **GOTO**.

26. Conversiones Implícitas: a resolver en trabajos prácticos 3/4

28. Comentarios de 1 línea: Comentarios que comiencen con "##" y terminen con el fin de línea.

Analizador Lexico

Decisiones de diseño e implementación

Durante la fase de diseño, se utilizó la herramienta YACC para la creación del autómata finito que representa el analizador léxico. Esta herramienta permite generar de manera eficiente el diagrama de transición de estados necesario para reconocer los diferentes tokens del lenguaje fuente. Java fue seleccionado como el lenguaje de programación principal debido a su robustez y capacidad para gestionar estructuras complejas, como las matrices de transición de estados. El analizador léxico se implementó de manera modular, con una clase principal llamada AnalizadorLexico, que utiliza diversas acciones semánticas encapsuladas en clases separadas (AS1, AS2, ..., ASE).

Además, se creó una tabla de símbolos encargada de almacenar los identificadores, constantes y cadenas presentes en el código fuente que se está compilando. Los símbolos en esta etapa solo contienen el lexema y el identificador correspondiente. Complementariamente, se implementaron dos estructuras adicionales: una tabla de palabras reservadas y una tabla de caracteres ASCII. La decisión de cuál de estas tablas utilizar, según el tipo de token que se esté procesando, es responsabilidad de las acciones semánticas. Dependiendo del estado actual y del carácter en curso, se ejecuta la acción semántica correspondiente, la cual no solo se encarga de armar correctamente los tokens, sino también de detectar posibles errores léxicos y almacenar los símbolos detectados en las tablas adecuadas.

En lo que respecta al análisis sintáctico, implementamos la gramática siguiendo las indicaciones propuestas por la cátedra. El compilador inicia su ejecución en el método main, donde se pasa el código fuente a compilar como un objeto de tipo Reader, el cual es procesado por el AnalizadorLexico. A continuación, se ejecuta la función run() del parser (analizador sintáctico), el cual solicita los tokens al léxico y construye las sentencias conforme se van detectando.

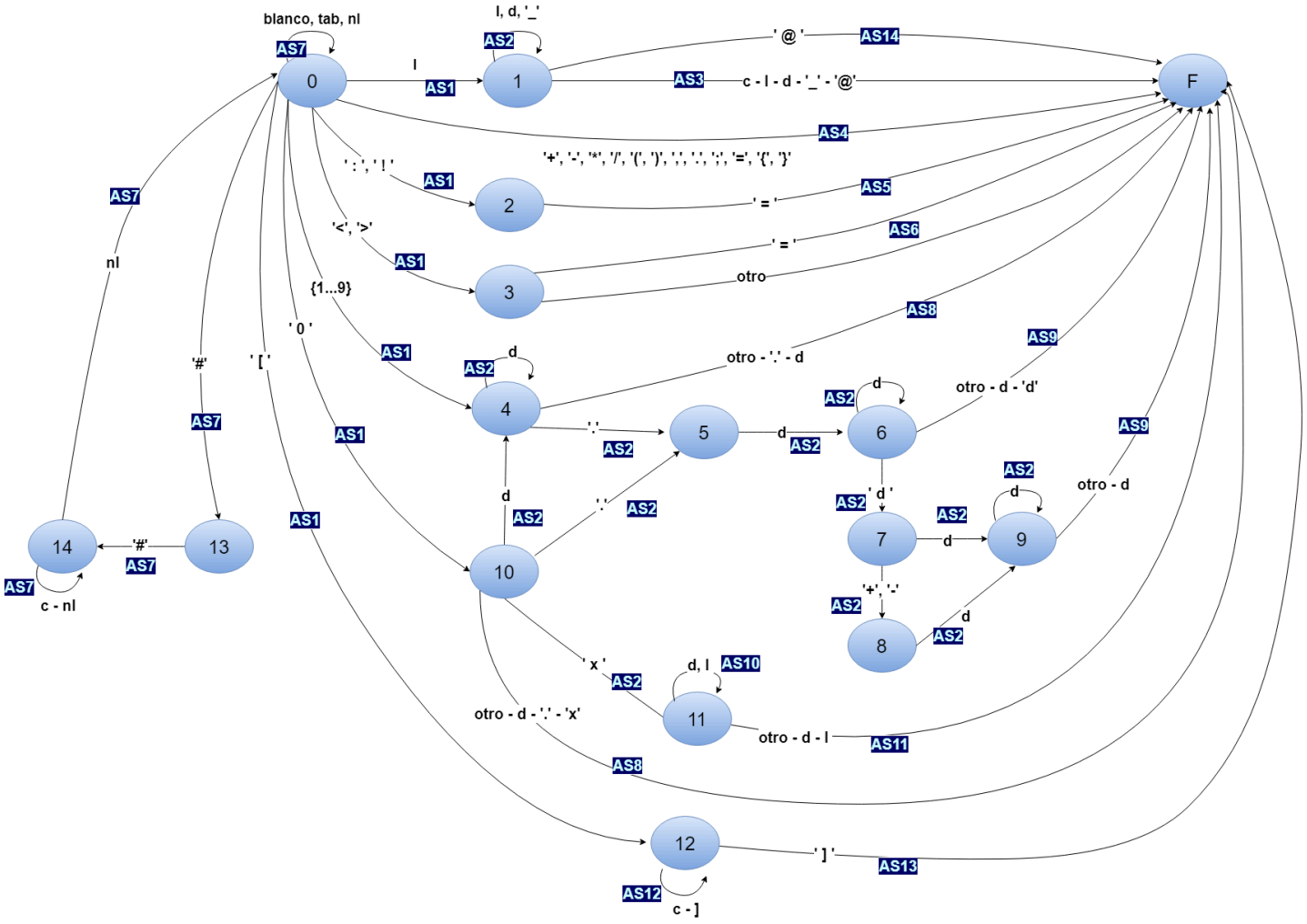
Una vez finalizada la compilación, el flujo retorna al main, donde se imprimen por pantalla todos los errores sintácticos encontrados. Además, se muestra una lista completa de los tokens reconocidos por el analizador léxico, así como los errores léxicos correspondientes.

Tanto las sentencias detectadas por el analizador sintáctico como los errores léxicos y sintácticos están acompañados de la línea de código en la que ocurrieron, utilizando una variable de control en el AnalizadorLexico, la cual se actualiza cada vez que se encuentra un nuevo salto de línea.

Además, para cada ejecución del programa con los distintos códigos de prueba se crea un archivo en blanco llamado TokensGenerados que se encuentra en la carpeta archivo donde se muestran todos los Tokens reconocidos para ese código en la línea correspondiente.

Es importante tener en cuenta que los archivos de prueba en formato **.txt** deben estar en formato Unix, ya que, de lo contrario, puede producirse un error en la primera línea. En algunos casos, los archivos pueden cambiar automáticamente al formato Windows, lo que genera incompatibilidades en la lectura del código.

Diagrama de transición de estados



Matriz de transición de estados

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
	L	d	x	bl / tab	nl	0	1-9	/	*	+	-	_	:	!	=	,	(	)	.	;	<	>	[	]	#	{	}	@	otro
0	1	1	1	0	0	10	4	F	F	F	F	-1	2	2	F	F	F	F	F	F	3	3	12	-1	13	F	F	-1	-1
1	1	1	1	F	F	1	1	F	F	F	F	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	F	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
4	F	F	F	F	F	4	4	F	F	F	F	F	F	F	F	F	F	F	5	F	F	F	F	F	F	F	F	F	F
5	-1	-1	-1	-1	-1	6	6	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	F	7	F	F	F	6	6	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
7	-1	-1	-1	-1	-1	9	9	-1	-1	8	8	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
8	-1	-1	-1	-1	-1	9	9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
9	F	F	F	F	F	9	9	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	11	F	F	4	4	F	F	F	F	F	F	F	F	F	F	F	5	F	F	F	F	F	F	F	F	F	F
11	11	11	11	F	F	11	11	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	F	12	12	12	12	12
13	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	14	-1	-1	-1	-1
14	14	14	14	14	0	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14

Matriz de acciones semánticas

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
	L	d	x	bl / tab	nl	0	1-9	/	*	+	-	_	:	!	=	,	(	)	.	;	<	>	[	]	#	{	}	@	otro
0	AS1	AS1	AS1	AS7	AS7	AS1	AS1	AS4	AS4	AS4	AS4	ASE	AS1	AS1	AS4	AS4	AS4	AS4	AS4	AS4	AS1	AS1	AS1	ASE	AS7	AS4	AS4	ASE	ASE
1	AS2	AS2	AS2	AS3	AS3	AS2	AS2	AS3	AS3	AS3	AS3	AS2	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS14	AS3
2	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	AS5	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE
3	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6
4	AS8	AS8	AS8	AS8	AS8	AS2	AS2	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS2	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8
5	ASE	ASE	ASE	ASE	ASE	AS2	AS2	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE
6	AS9	AS2	AS9	AS9	AS9	AS2	AS2	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9
7	ASE	ASE	ASE	ASE	ASE	AS2	AS2	ASE	ASE	AS2	AS2	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE
8	ASE	ASE	ASE	ASE	ASE	AS2	AS2	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE
9	AS9	AS9	AS9	AS9	AS9	AS2	AS2	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9	AS9
10	AS8	AS8	AS2	AS8	AS8	AS2	AS2	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS2	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8
11	AS10	AS10	AS10	AS11	AS11	AS2	AS2	AS11	AS11	AS11	AS11	AS11	AS11	AS11	AS11	AS11	AS11	AS11	AS11	AS11	AS11	AS11	AS11	AS11	AS11	AS11	AS11	AS11	AS11
12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS12	AS13	AS12	AS12	AS12	AS12	AS12
13	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	AS7	ASE	ASE	ASE	ASE
14	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7

Para visualizar las matrices de manera más clara, puede acceder al siguiente enlace: https://docs.google.com/spreadsheets/d/16LTJHriGKRTt4uFPQj089ZImJ7ki_CZcKiddSDpfWC8/edit?usp=sharing

El proceso de creación de las matrices se basó en el diseño del autómata finito. Primero, se creó la **matriz de transición de estados**, donde cada fila representa el estado del autómata y cada columna un símbolo de entrada. El valor en cada celda indica el próximo estado al que se debe mover el autómata según el símbolo leído, ya sea cambiando de estado, permaneciendo en el mismo o llegando a un estado final.

Posteriormente, se construyó la **matriz de acciones semánticas**, la cual trabaja en paralelo con la matriz de transición. Esta matriz define qué acción semántica debe ejecutarse en cada transición, como construir un token, almacenarlo en la tabla de símbolos, o detectar errores léxicos. Las acciones se seleccionan en función del estado actual y el símbolo procesado.

Ambas matrices, al trabajar juntas, permiten al analizador léxico avanzar a través del código fuente de manera eficiente y realizar las acciones necesarias para la correcta construcción y clasificación de tokens.

En la implementación del compilador, se utilizaron dos archivos de texto en Java: uno para la matriz de transición de estados y otro para la matriz de acciones semánticas. El compilador maneja estos archivos en la clase AnalizadorLexico.

La clase define dos matrices:

- transicion_estados: es la matriz que determina a qué estado debe moverse el autómata léxico basándose en el estado actual y el carácter leído.
- acciones_semanticas: es la matriz que determina qué acción semántica debe ejecutarse en cada transición.

Ambas matrices se leen desde archivos de texto externos utilizando los métodos leerMatrizDeTransicion y leerAccionesSemanticas. Estos métodos abren los archivos correspondientes, recorren cada línea y rellenan las matrices fila por fila.

Una vez que las matrices de transición de estados y acciones semánticas están cargadas en memoria, utiliza un estado inicial (estado_actual) que se actualiza a medida que se leen los caracteres del código fuente. Para cada carácter, se consulta la matriz de transición de estados para determinar el próximo estado, mientras que la matriz de acciones semánticas define la operación a realizar, como la creación de un token o la detección de errores.

Durante la ejecución, el analizador va construyendo tokens dinámicamente a través del atributo token_actual.

Acciones semánticas

AS1: Inicializa string y añade el carácter leído.

AS2: Lee carácter y lo concatena al string.

AS3: Verifica si es una palabra reservada o verifica y/o agrega en la tabla de símbolos, si supera la longitud máxima se trunca, y devuelve el token.

AS4: Reconoce literal y devuelve token.

AS5: Lee carácter y lo añade al string, verifica y/o agrega en palabras reservadas y devuelve el token.

AS6: Verifica si el carácter leído es un '='. Si es así, busca en las palabras reservadas para determinar si se está formando el operador `<=` o `>=` y retorna el token correspondiente. Si no se encuentra un '=', devuelve el último carácter leído, que será `>` o `<`, según corresponda.

AS7: Lee carácter sin agregarlo al string.

AS8: Verifica que el entero no exceda el rango permitido, verifica y/o agrega en la tabla de símbolos y devuelve el token.

AS9: Verifica que el punto flotante no supere el rango permitido, verifica y/o agrega en la tabla de símbolos y devuelve el token.

AS10: Lee carácter y lo concatena al string, si es una letra verifica que no supere el rango permitido (A-F).

AS11: Verifica que sea un rango permitido para constante hexadecimal en integer(-0x8000 a 0x7FFF). Lo agrega en la tabla de símbolos y retorna el token.

AS12: Lee carácter y lo concatena al string. Si es un salto de línea, se actualiza la línea actual.

AS13: Lee y agrega el último carácter a la cadena. Agrega el string a la tabla de símbolos sin saltos de línea y devuelve el token.

AS14: Lee carácter y lo concatena al string, si supera la longitud máxima se trunca. Verifica y/o agrega en la tabla de símbolos y devuelve el token. (para etiqueta)

ASE: Retorna error(-1).

Errores léxicos considerados

Errores Lexicos

- El identificador es demasiado largo, se lo cortó a la longitud máxima de 15 caracteres.
- No se reconoce el carácter introducido.
- TOKEN NO VALIDO: (token_actual)
- El entero ingresado se encuentra fuera de rango, será cambiado por el entero más cercano permitido
    - Error: Linea + (linea actual): Supera el rango
- El número flotante está fuera del rango permitido, se ajustará al valor más cercano permitido.
    - El número es demasiado grande. Se lo ha reemplazado por el más cercano posible:
    - El número es demasiado chico. Se lo ha reemplazado por el más cercano posible:
    - El número flotante introducido no es válido.
- Caracter no valido
- El valor está fuera del rango permitido para un HEXADECIMAL.
- La etiqueta supera la longitud máxima de 15 caracteres,se lo cortó a la longitud máxima.
- Error no descrito en declaración léxica. Verifique el formato de los caracteres ingresados y del salto de línea (LF)

Analizador Sintáctico

Temas Particulares

Tema 11: Subtipos
Se debe incorporar la posibilidad de declarar tipos como subrangos de los tipos básicos aceptados por el lenguaje. En la declaración se incluirá el nombre del nuevo tipo, el tipo en el que se basa, y el rango de valores aceptado por ese tipo
Por ejemplo:

typedef enterito := **integer**{-10,10}; //declara un tipo subrango de enteros
typedef flotantito := **single**{-1.2,3.5}; //declara un tipo subrango de single

Se podrán declarar variables de los tipos definidos, que se utilizarán del mismo modo que cualquier variable. Ejemplos:

```
enterito a, b, c;
flotantito x, y, z;
a := 1;
z := x + y;
```

Tema 13: WHILE

WHILE (<condicion>) <bloque_de_sentencias_ejecutables> ;

<condición> tendrá la misma definición que la condición de las sentencias de selección.
<bloque_de_sentencias_ejecutables> podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por begin end.

Tema 18: Asignaciones múltiples
Se deben reconocer asignaciones múltiples, que permitan una lista de variables, separadas por coma (“,”), del lado izquierdo de la asignación, y una lista de expresiones, separadas por coma “,”), del lado derecho.
Por ejemplo:

```
a,b,c := 1+d, e, f+5;
x,y := 1, f1(3), w;
i, j , k := a*3 , b;
```

Tema 21: pair
Incorporar la posibilidad de declarar tipos pair de un tipo determinado (considerando los tipos asignados al grupo).
Por ejemplo:

typedef pair <**integer**> pint; //declara un tipo par de enteros

Se podrán declarar variables del tipo definido (en el ejemplo pint), y los componentes de cada variable se referenciarán con el nombre de la variable y la posición de la componente entre corchetes, considerando 1 y 2 para la primera y segunda componente, respectivamente.
Por ejemplo:

```
pint p1,p2,p3; // declaración de las variables p1 a p3 del tipo pint
p1{1} := 8; // se asigna 8 a la primera componente de la variable p1
p2 := p1; // se asigna la variable p2 con la variable p1
x := p2{1}; // se asigna x con la primera componente de la variable p2
```

Tema 23: goto

Permitir, como sentencia ejecutable, la sentencia goto <etiqueta>, donde la etiqueta será un identificador seguido de dos puntos (":"). Esta etiqueta podrá aparecer en cualquier lugar del código, como una sentencia más.

Ejemplo de código para esta funcionalidad:

```
…
for (i := 1; i < 10; up 1)
begin
        if (...)
        then
                goto afuera@;
        else
                z := z + I;
        End_if;
end;
afuera@
```

Tema 26: Conversiones Implícitas

Se explicará y resolverá en trabajos prácticos 3 y 4.

Descripción del proceso de desarrollo

El desarrollo del analizador sintáctico comenzó con la definición de la gramática. Primero, analizamos los temas generales del lenguaje objetivo y los temas específicos asignados por la cátedra. Con esta base, construimos una gramática que cubría tanto las reglas fundamentales del lenguaje como las particularidades.

Una vez terminada esta fase, procedimos a generar el parser utilizando YAAC. Durante este proceso, surgieron conflictos de tipo shift-reduce y reduce-reduce, los cuales YAAC detectó al intentar generar el autómata de análisis sintáctico. Para resolver estos conflictos, analizamos cuidadosamente las reglas gramaticales que podrían estar causando ambigüedad y las modificamos. Este proceso incluyó reestructurar ciertas producciones y agregar precedencia en algunas reglas hasta que logramos eliminar todos los conflictos.

Los principales problemas surgieron en torno a las ambigüedades presentes en la gramática inicial. Los conflictos shift-reduce se resolvieron dando prioridad a las reglas de producción que requerían un desplazamiento (shift) cuando era necesario, mientras que los conflictos reduce-reduce se solucionaron al ajustar las reglas con operadores ambiguos o eliminando producciones redundantes. Estas modificaciones nos permitieron generar un parser libre de errores de ambigüedad.

Durante el proceso de desarrollo y testing, implementamos un mecanismo de manejo de errores que permitía al analizador sintáctico continuar el análisis aun cuando se detectaban errores en el código fuente. Para esto, agregamos reglas especiales en la gramática que permitían la recuperación y reportaban los errores junto con la línea en la que ocurrían. Esto fue especialmente útil para manejar errores comunes como tokens faltantes o mal estructurados.

Una vez que resolvimos los conflictos, comenzamos a realizar pruebas utilizando códigos de ejemplo. Durante estas pruebas, verificamos que cada parte de la gramática funcionara correctamente. Si el análisis sintáctico fallaba, revisábamos las reglas gramaticales y las ajustábamos para corregir el problema. Después de confirmar que la gramática manejaba correctamente los casos válidos, agregamos pruebas de códigos con errores para validar el comportamiento del manejador de errores. Este proceso iterativo permitió afinar tanto la gramática como el manejo de errores del compilador.

Lista de no terminales

- **program**: Representa todo el programa.
- **programa**: Contiene el nombre del programa y un bloque de sentencias.
- **nombre_programa**: Define el identificador del nombre del programa.
- **bloque_sentencias_programa**: Contiene una o más sentencias de programa.
- **sentencia_programa**: Abarca tipos de sentencias dentro del programa: declarativas, ejecutables y etiquetas.
- **sentencias_declarativas**: Incluye varias declaraciones: variables, tipos, funciones
- **declaracion_variable**: Define la declaración de variables con su tipo y lista de variables.
- **declaracion_funcion**: Define la declaración de funciones, incluyendo su tipo de retorno, nombre, parámetros y cuerpo.
- **parametro**: Representa los parámetros de una función, incluyendo su tipo y nombre.
- **cuerpo_funcion**: Contiene las sentencias dentro del cuerpo de la función y la sentencia de retorno.
- **sentencia_retorno**: Define las sentencias de retorno con valor de retorno.
- **invocacion_funcion**: Representa la invocación de una función con su parámetro.
- **list_var**: Define una lista de variables.
- **tipo**: Define los tipos de datos permitidos en la declaración de variables y funciones.
- **sentencias_ejecutable**s: Conjunto de sentencias ejecutables, como asignación, selección, salida, bucles y saltos.
- **sentencia_asignacion**: Asignación de variables a expresiones.
- **list_expresion**: Lista de expresiones aritméticas.
- **sentencia_seleccio**n: Sentencia de control IF-THEN-ELSE.
- **bloque_sentencias_del**: Bloque de sentencias delimitado por BEGIN y END.
- **bloque_ejecutable**: Bloque de sentencias ejecutables.
- **condicion**: Condición de comparación entre expresiones aritméticas.
- **comparador**: Define los operadores de comparación, usados en condiciones para comparar valores.
- **sentencia_ou**t: Sentencia de salida que imprime una cadena o expresión aritmética.
- **declaracion_tipo_subrango**: Declaración de un tipo subrango usando TYPEDEF.
- **rango**: Define el rango de valores permitido entre llaves {}.
- **sentencia_while**: Sentencia de control WHILE para iteraciones.
- d**eclaracion_tipo_pair**: Declaración de un tipo PAIR usando TYPEDEF.
- **sentencia_goto**: Sentencia GOTO para saltar a una etiqueta específica.
- **llamado_etiqueta**: Define la estructura de un salto GOTO a una etiqueta.
- **etiqueta**: Define una etiqueta en el código.
- **expr_aritmetic**: Expresiones aritméticas que pueden incluir operaciones de suma y resta.
- **termino**: Términos aritméticos, incluyendo multiplicación, división y factores.
- **factor**: Factores de una expresión, que pueden ser identificadores, constantes, o identificadores con un valor constante.
- **const**: Define una constante, que puede ser positiva o negativa.

Lista de errores sintácticos considerados

Errores Sintanticos

- programa:
    - Error en las sentencias de ejecución.
    - Falta de delimitador de programa.
    - Error sintáctico al compilar.
    - Falta de nombre de programa.
    - Falta de sentencias de ejecución.
- sentencias_declarativas:
    - Falta de ‘;’ después de las declaraciones.
- declaracion_funcion:
    - Falta de nombre en la función.
    - Falta de parámetro.
    - Falta de tipo.
- parametro:
    - Falta de identificador luego del tipo.
    - Falta de tipo de parámetro formal en declaración de función.
- cuerpo_funcion:
    - Falta de sentencia RET.
- sentencia_retorno:
    - Falta RET.
    - Falta de paréntesis.
    - Falta de ‘;’ después del retorno.
- list_var:
    - Falta de , en la declaración de variables.
- sentencias_ejecutables:
    - Falta ; después de la sentencia.
- sentencia_asignacion:
    - Error en la expresión de la asignación.
- list_expresion:
    - Falta de , en la lista de expresiones.
- sentencia_seleccion:
    - Falta de paréntesis en la condición.
    - Falta de END_IF.
    - Falta de contenido en bloque THEN/ELSE.
- bloque_sentencias_del:
    - Falta de contenido en bloque.
    - Falta el BEGIN o END.
- condicion:
    - Falta de expresión aritmética en comparación.
- sentencia_out:

    - Falta parámetro en sentencia OUTF.
    - Falta el paréntesis de cierre/apertura.
- declaracion_tipo_subrango:
    - Falta el nombre del tipo definido.
    - Falta tipo base.
    - Falta de {} en el rango.
- sentencia_while:
    - Falta de paréntesis en la iteración.
    - Falta WHILE.
- declaracion_tipo_pair:
    - Falta PAIR.
    - Falta de <> en la declaración de PAIR.
    - Falta identificador al final de la declaración.
    - Falta tipo en la declaración de PAIR.
- sentencia_goto:
    - Falta etiqueta en la sentencia GOTO.
- expr_aritmetic:
    - Falta de operando en expresión.

Conclusiones

El desarrollo de este compilador en Java utilizando YACC ha sido un proceso que nos permitió aplicar de manera práctica los conceptos teóricos estudiados. La implementación del Analizador Léxico y del Analizador Sintáctico fue fundamental para comprender cómo los lenguajes de programación son procesados por una máquina. A través de la creación del autómata finito y las tablas de transición y acciones semánticas, logramos descomponer y analizar el código fuente de manera efectiva, identificando tokens, estructuras válidas, y detectando errores léxicos y sintácticos.

La implementación modular, tanto del analizador léxico como del sintáctico, nos permitió resolver problemas de ambigüedad, mejorar el rendimiento del compilador, y optimizar el manejo de errores. Adicionalmente, el proceso de desarrollo y prueba nos brindó una comprensión sobre cómo diseñar gramáticas para que sean claras y precisas, minimizando los conflictos shift-reduce y reduce-reduce.