Trabajo Práctico Especial
Diseño de compiladores I

Número de grupo: 12

Integrantes:
- Apraiz Thomás: tominola99@gmail.com
- Martinez Leanes Manuel: manumleanes@gmail.com
- Lujan Nicolas: Nico.L.2014@hotmail.com

Profesor asignado:
- José Fernández León

Temas asignados: 3 7 (9) 10* 13 18 20 (21) 22* 24 27

Índice

Introducción

En esta primera etapa del trabajo práctico especial de la materia “Diseño de compiladores I” se desarrolló la parte léxica y sintáctica (con sus respectivos analizadores) de un compilador basado en características especiales de un lenguaje dado como consigna por la cátedra de la materia. Dicho compilador está implementado mediante una arquitectura monolítica en la cuál el parser es el main, “pide” tokens al Analizador Léxico y “entrega” reglas al Generador de Código.
El trabajo fue implementado en el lenguaje Java para la construcción del analizador léxico y sintáctico. Por otro lado se utilizó Yacc para la implementación del parser. En el siguiente informe se contemplan todas las técnicas y aclaraciones dadas por la cátedra para lograr un correcto funcionamiento de ambos analizadores.

Temas Particulares Asignados

- Tema 3. Enteros (16 bits): Constantes enteras con valores entre –2 15 y 215 – 1. Se debe incorporar a la lista de palabras reservadas la palabra i16
- Tema 7. Punto Flotante de 32 bits: Números reales con signo y parte exponencial. El exponente comienza con la letra F (mayúscula) y el signo es opcional. La ausencia de signo, implica un exponente positivo. La parte exponencial puede estar ausente. Puede estar ausente la parte entera, o la parte decimal, pero no ambas. El „." es obligatorio. Ejemplos válidos: 1. .6 -1.2 3.F–5 2.F+34 2.5F-1 15. 0. 1.2F10. Considerar el rango 1.17549435F-38 < x < 3.40282347F+38 -3.40282347F+38 < x < -1.17549435F-38 0.0
- Tema 10: Cláusulas de compilación
  Incorporar, a las sentencias declarativas, la posibilidad de definir constantes.
- Tema 13. Do until con continue: Incorporar a la lista de palabras reservadas las palabras do, until y continue.
- Tema 18. Continúe con etiquetado.
- Tema 20. Sentencia de control como expresión.
- Tema 22. Conversiones Implícitas
- Tema 24. Comentarios de 1 línea: Comentarios que comienzan con “<<” y terminan con el fin de línea.
- Tema 27. Cadenas multilínea: Cadenas de caracteres que comiencen y terminen con “ ‘ ” . Estas cadenas pueden ocupar más de una línea, y en dicho caso, al final de cada línea, excepto la última debe aparecer una barra “ / ”. (En la Tabla de símbolos se guardará la cadena sin las barras, y sin los saltos de línea).
  Ejemplo: ‘¡Hola /
  mundo!’

Analizador Léxico

Consignas generales

Desarrollar un Analizador Léxico que reconozca los siguientes tokens:

- Identificadores cuyos nombres pueden tener hasta 25 caracteres de longitud. El primero debe ser una letra, y el resto pueden ser letras, dígitos y "_". Los identificadores con longitud mayor serán truncados y esto se informará cómo Warning. Las letras utilizadas en los nombres de identificadores pueden ser minúsculas y/o mayúsculas.
- Constantes correspondientes al tema particular asignado a cada grupo. Nota: Para aquellos tipos de datos que pueden llevar signo, la distinción del uso del símbolo "-" como operador aritmético o signo de una constante, se postergará hasta el trabajo práctico Nro. 2.
- Operadores aritméticos: "+", "-" , "*", "/" agregando lo que corresponda al tema particular.
- Operador de asignación: "=:"
- Comparadores: ">=", "<=", ">", "<", "=", "=!" "(", ")","{", "}", "," y ";"
- Cadenas de caracteres correspondientes al tema particular de cada grupo.
- Palabras reservadas (en minúsculas): if, then, else, end-if, out, fun, return, break y demás símbolos / tokens indicados en los temas particulares asignados al grupo.

El Analizador Léxico debe eliminar de la entrada (reconocer, pero no informar como tokens al Analizador Sintáctico), los siguientes elementos:

- Comentarios correspondientes al tema particular de cada grupo
- Caracteres en blanco, tabulaciones y saltos de línea, que pueden aparecer en cualquier lugar de una sentencia.

Decisiones de diseño e implementación

Para la realización de esta primera etapa se utilizó el lenguaje de programación Java. Se creó el Analizador Léxico el cuál se encarga de leer el programa fuente y de retornar un entero cada vez que reconoce un token por medio del método “yylex”. Además cada vez que lee un carácter el Analizador Léxico cambia de estado (mediante una matriz de transición de estados) y ejecuta la acción semántica (Almacenada en una matriz de Acciones Semánticas) asociada a ese cambio de estado. El reconocimiento de un token por parte del Analizador Léxico se realiza mediante un diagrama de transición de estados (autómata finito) el cuál fué implementado con dicha matriz de transición de estados de tamaño NxM donde N es la cantidad de estados y M la cantidad de símbolos que se pueden reconocer. De la misma manera se creó la matriz de acciones semánticas la cuál almacena las distintas acciones que se deben tomar ante un cambio de estado (arcos del autómata). Ambas matrices son atributos que posee la clase Analizador Léxico y se pueden cargar dinámicamente por medio de métodos.

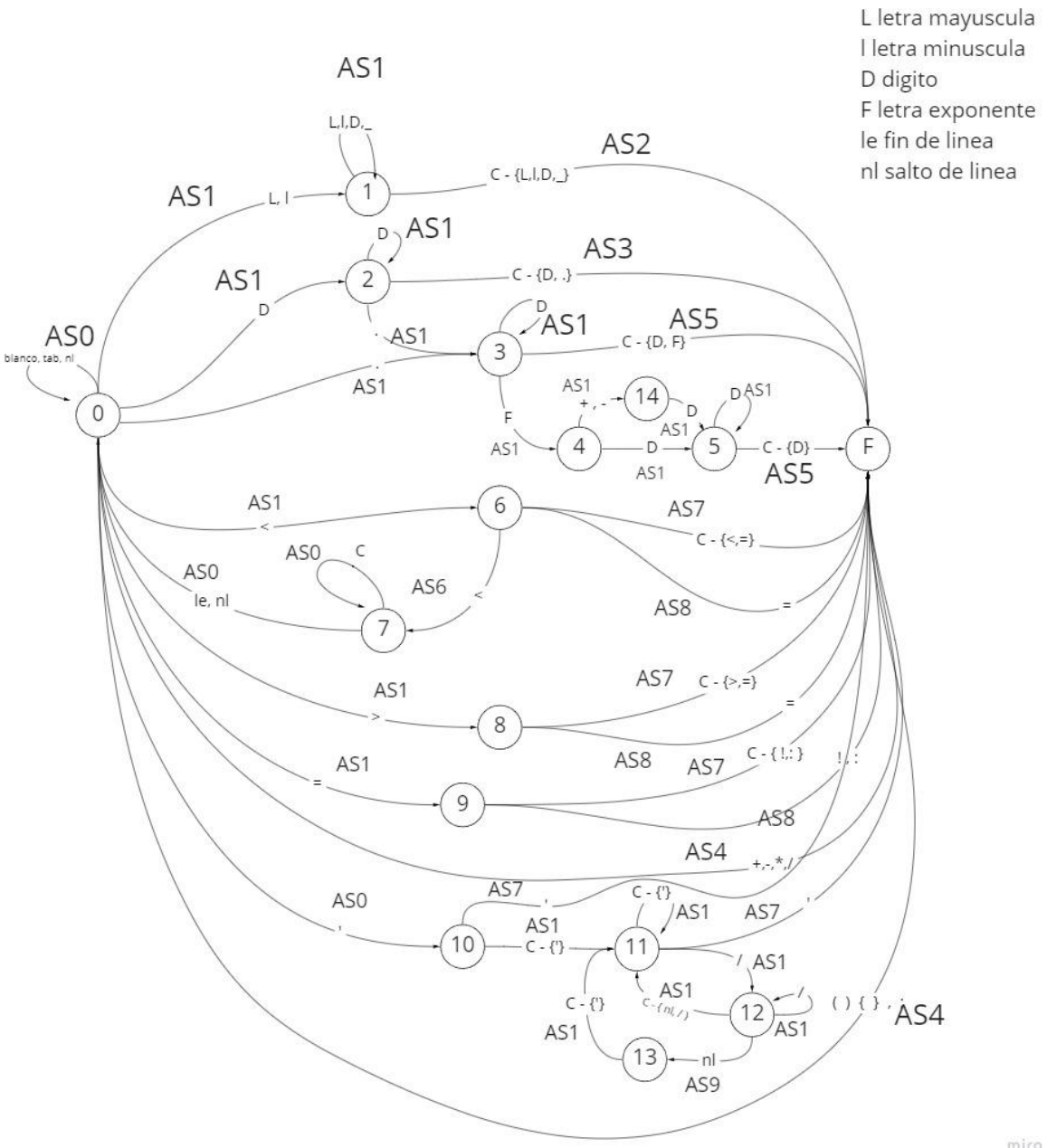
Por otro lado se implementó la clase **Tabla de Símbolos**, la cuál es una estructura de datos que almacena un registro para cada símbolo utilizado en el código fuente, con campos que contienen información relevante para cada símbolo (**Atributos**). Sólo se registran en la Tabla aquellos símbolos para los cuales pueden existir distintos lexemas para un mismo token. Dicha tabla implementó con un HashMap donde la clave es un String (Lexema del token) y el valor es el **Atributo**, es decir, otra clase en java la cuál es encargada de almacenar distintos tipos de información (Ej: Lexema, nro. de línea en que aparece el token, tipo, etc.) que se usarán en etapas posteriores (Generación de código). También se construyó una clase de **Tabla de Palabras Reservadas** la cuál almacena las distintas palabras reservadas del lenguaje y es usada tanto como por el analizador léxico y el sintáctico.

Con respecto a las **Acciones Semánticas** consideramos implementarla mediante una clase abstracta con el método abstracto de ejecutar donde cada acción semántica particular extiende de Acción Semántica y sobreescribe el método para realizar distintas acciones. Dicho método recibe como parámetros un *StringBuilder* que representa al token actual, y un *Reader* o lector encargado de leer el código fuente del programa.

Errores léxicos considerados

El Analizador Léxico puede producir ciertos warnings y errores a la hora de compilar el código fuente. Por ejemplo en el caso de los identificadores cuyo tamaño supere los 25 caracteres será truncado e informado como warning. Por el lado de los errores el Analizador Léxico puede detectar caracteres inválidos en el mapeo de los símbolos de entrada, las constantes fuera de rango y cuando el carácter de entrada no coincide con ninguno de los arcos de salida en un estado del autómata (Por ejemplo si se reconoce una F como exponente de un número flotante lo debe seguir obligatoriamente un dígito, sino da error).

Diagrama de transición de estados



Matriz de transición de estados

	L	l	D	_	Blanco	;	,	F	.	<	>	"="	!	:	'	+	-	*	/	{	}	(	)	Tab	nl	otro
0	1	1	2		0	F	F	1	3	6	8	9	F	F	10	F	F	F	F	F	F	F	F	0	0	
1	1	1	1	1	F	F	F	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
2	F	F	2	F	F	F	F	F	3	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
3	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			5													14	14									
5	F	F	5	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
6	F	F	F	F	F	F	F	F	F	7	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	0	7
8	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
9	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
10	11	11	11	11	11	11	11	11	11	11	11	11	11	11	F	11	11	11	11	11	11	11	11	11	11	11
11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	F	11	11	11	12	11	11	11	11	11	11	11
12	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	12	11	11	11	11	11	13	11
13	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
14			5																							

Aclaracion: los casilleros vacios indican que es un error.

Matriz de Acciones Semánticas

	L	l	D	_	Blanco	;	,	F	.	<	>	"="	!	:	'	+	-	*	/	{	}	(	)	Tab	nl	otro
0	AS1	AS1	AS1	AS1	AS0	AS4	AS4	AS1	AS1	AS1	AS1	AS1	ASE	ASE	AS0	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS0	AS0	ASE
1	AS1	AS1	AS1	AS1	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2
2	AS3	AS3	AS1	AS3	AS3	AS3	AS3	AS3	AS1	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3
3	AS5	AS5	AS1	AS5	AS5	AS5	AS5	AS1	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5
4	ASE	ASE	AS1	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	AS1	AS1	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE
5	AS5	AS5	AS1	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5
6	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS6	AS7	AS8	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7
7	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0
8	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS8	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7
9	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS8	AS8	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7	AS7
10	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS7	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	ASE
11	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS7	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	ASE
12	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS9	ASE
13	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	ASE	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	ASE
14	ASE	ASE	AS1	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE

Acciones Semánticas implementadas

AS0: leer siguiente carácter.
AS1: concatenar el carácter leído.
AS2:
-Devolver a la entrada el último carácter leído
-Verificar rango (< 25 caracteres)
-Buscar en la TPR:
  -Si está, devolver token de PR
  -Si no está:
    -Buscar en la TS:
      -Si está, devolver ID + Punt TS
      -Si no está: -Alta en TS -Devolver ID + Punt TS
AS3:
-Agregar lexema a la tabla de símbolos (si no está), verificar rango de entero y devolver identificador del token.
AS4: Encargada de leer los símbolos: '+' '-' '/' '(' ')' ',' ';' '*' , reconocer el literal, y devolver Token del mismo.
AS5:
-Agregar lexema a la tabla de símbolos, verificar rango de flotante y devolver identificador del token.
AS6:
Esta acción está destinada a los comentarios. Suprime el token leído hasta el momento, y lee el siguiente carácter. Ya que los comentarios no debemos enviarlos al Parser, no devolvemos el token correspondiente solo lo ignoramos.
AS7:
Devolver el identificador del token correspondiente
AS8:
Concatenar el carácter y devolver el token correspondiente
AS9:
Borra el último carácter leído y lee el siguiente
ASE:
Informa que hay error

Analizador Sintatico

Consigna General

Construir un Parser (Analizador Sintáctico) que invoque al Analizador Léxico creado en el Trabajo Práctico Nº 1, y que reconozca un lenguaje con las características dadas.
a) Utilizar YACC u otra herramienta similar para construir el Parser.
b) Adaptar el Analizador Léxico del Trabajo Práctico 1 para convertirlo en el método o función int yylex() (o el nombre que el Parser generado requiera). Tener en cuenta que el léxico deberá devolver al parser, en cada invocación, un token. Para los identificadores, constantes y cadenas, deberá devolver además, la referencia a la entrada de la Tabla de Símbolos donde se ha registrado dicho símbolo, utilizando yylval para hacerlo.
c) Para aquellos tipos de datos que permitan valores negativos (i8, i16, i32, f32, f64) durante el Análisis Sintáctico se deberán detectar constantes negativas, modificando la tabla de símbolos según corresponda. Será necesario volver a controlar el rango de las constantes, ya que un valor aceptado para una constante por el Analizador Léxico, que desconoce su signo, podría estar fuera de rango si la constante es positiva. Ejemplo: Las constantes de tipo i16 pueden tomar valores desde –32768 a 32767. El Léxico aceptará la constante 32768 como válida, pero si se trata de una constante positiva, estará fuera de rango.
d) Cuando se detecte un error, la compilación debe continuar.
e) Conflictos: Eliminar TODOS LOS CONFLICTOS SHIFT-REDUCE Y REDUCE-REDUCE que se presenten al generar el Parser.

Descripcion del proceso de desarrollo

Para realizar el trabajo práctico 2 lo primero que se hizo fué plantear la gramática del lenguaje dado por la cátedra. Las cuales se analizaron, revisaron y corrigieron en varias oportunidades para cumplir con los requisitos impuestos por el lenguaje. También se intentó formular reglas lo más claras y legibles posibles. Se tuvo que modificar y reescribir la gramática para que las sentencias declarativas pueden aparecer en cualquier lugar del código fuente, exceptuando los bloques de las sentencias de control. Por otro lado en el caso de las declaraciones de funciones el número máximo de parámetros permitidos es 2, y puede no haber parámetros, por lo tanto, tuvimos que hacer este chequeo durante el Análisis Sintáctico creando ciertas reglas para controlar esta restricción.

Problemas surgidos y soluciones adoptadas en el proceso

Algunos problemas surgidos en la implementación de la gramática se dieron a la hora de los caracteres que encierran sentencias “{” “}” y el uso “;” al final de una sentencia por la recursión y las sentencias anidadas.
Por otro lado, cuando ya se implementó la gramática, surgieron ciertos problemas de reduce-reduce y shift-reduce, por ejemplo, en el caso de la sentencia if y de expresiones gramaticales por lo que se optó por modificar la gramática para solucionar estos inconvenientes. En el caso de las expresiones no se deben permitir anidamientos con paréntesis por lo que había que adaptar la gramática para la precedencia y asociatividad de las operaciones.
Además se debió incorporar al léxico el símbolo “:”, la palabra reservada end_if, la palabra reservada const

Manejo de errores

El manejo de errores esta implementado mediante codigo asociado a algunas reglas de la gramatica e imprimiendo por pantalla los errores encontrados. Posteriormente, implementaremos los errores mediante clases y guardaremos los mismos junto a la linea en la que se produjo.

Lista de no terminales usados en la gramática

- programa: Es la regla principal de la gramática (%start).
- nombre_prog: Es la regla que nos indica el nombre del programa.
- cuerpo_prog: Regla que sirve para la construcción del programa.
- bloque: Regla que permite definir bloques del programa.
- sentencia: Permite definir las distintas sentencias que puede soportar el parser.
- declaracion: Sirve para escribir distintos tipos de sentencias declarativas.
- declaracion_const: Regla que permite reconocer constantes (tema 10).
- list_const: Permite declarar recursivamente constantes.
- declaracion_var: Regla utilizada para la declaracion de variables.
- lista_de_variables Permite declarar recursivamente variables.
- variable: Permite declarar variables con un ID
- tipo: Regla que permite definir tipos I16 o F32
- declaracion_func: Esta regla sirve para declarar funciones en el programa.
- header_func: Se utiliza para escribir correctamente el encabezado de una funcion.
- lista_parametros_funcion: Permite declarar los parametros de una función.
- cuerpo_func: Regla que sirve para reconocer el cuerpo de una función.
- nombre_func: Permite identificar el nombre de una función (ID).
- ejecucion: Regla que nos permite declarar distintos tipos de sentencias de ejecucion.
- asignacion:
- expresion_aritmetica
- termino
- factor

- lista_inv_func
- seleccion
- then_seleccion
- else_seleccion
- condicion
- operador
- impresion
- estruct_do_until
- bloque_do_until
- etiqueta
- sentencia_ctr_expr
- bloque_do_until_expr

Conclusiones

Con el desarrollo de los trabajos practicos 1 y 2, pudimos aprender el funcionamiento e implementacion de un analizador lexico y un analizador sintactico, como entre ellos se comunican para llevar a cabo la compilacion. En nuestro caso, aprendimos como funciona un compilador con arquitectura monolitica, donde el parser es el main y le pide tokens al lexico, para este poder construir la lista de reglas de la gramatica.