Universidad Nacional de Río Cuarto

Taller de Diseño de Software

(Código 3306)

Compilador: TDS25

Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales

Autores
Joaquín Mezzano
Tomás Rodeghiero

Fecha de entrega: 15 de Septiembre de 2025

Índice

1. Análizador Sintáctico y Léxico Inicio: (Scanner y Parser)

1.1. Introducción

Este documento presenta la primera etapa del desarrollo del compilador para el lenguaje **TDS25**, correspondiente a la implementación del Analizador Léxico (Scanner) y el Analizador Sintáctico (Parser). El proyecto forma parte de la materia **Taller de Diseño de Software (Código 3306)** de la **Universidad Nacional de Río Cuarto**.

El lenguaje TDS25 es un lenguaje imperativo simple, similar a C o Pascal, que incluye variables de tipos básicos (entero y booleano), funciones, estructuras de control condicionales e iterativas, y un mecanismo para invocar métodos de librerías externas.

Esta etapa inicial del compilador se enfoca en la transformación del código fuente en tokens válidos y la verificación de la estructura sintáctica del programa según la gramática especificada para TDS25.

1.2. División del Trabajo

Nuestro grupo está compuesto por los siguientes integrantes:

- **Joaquín Mezzano**
- **Tomás Rodeghiero**

Dado nuestro tamaño reducido del grupo, el trabajo se realizó de manera colaborativa.

1.3. Decisiones de Diseño y Asunciones

Decisiones del Analizador Léxico

- **Manejo de enteros negativos**: Decidimos reconocer literales enteros negativos directamente en el lexer mediante el patrón `[-]?{DIGIT}+`, lo que simplifica el parser y evita ambigüedades con el operador menos unario.
- **Validación de rango**: Implementaos validación del rango de enteros de 32 bits con signo (-2147483648 a 2147483647) directamente en el analizador léxico, reportando errores cuando se exceden estos límites.
- **Soporte númerico**: Incluimos `integer` como palabra reservada del tipo entero, con el objetivo de lograr una mayor flexibilidad númerica en los ejemplos de prueba.
- **Manejo de comentarios**: Implementamos expresiones regulares para reconocer tanto comentarios de línea (`//`) como comentarios multilínea (`/* */`), descartándolos apropiadamente.

Decisiones del Analizador Sintáctico

- **Precedencia de operadores**: Definimos la precedencia de operadores siguiendo la especificación del lenguaje, con el menos unario y la negación lógica teniendo la mayor precedencia.
- **Asociatividad**: Establecimos asociatividad por izquierda para la mayoría de operadores binarios, siguiendo las convenciones estándar de lenguajes imperativos.
- **Mensajes de error**: Habilitamos la generación de mensajes de error detallados con `%define parse.error verbose` para facilitar la depuración.

1.4. Descripción del Diseño

Arquitectura General
El compilador sigue una arquitectura tradicional de dos fases secuenciales:

1. **Análisis Léxico**: Convierte el flujo de caracteres del código fuente en una secuencia de tokens válidos.
2. **Análisis Sintáctico**: Verifica que la secuencia de tokens cumple con la gramática del lenguaje TDS25.

Analizador Léxico (lexico.l)
El analizador léxico fue implementado utilizando Flex y contiene:

- **Definiciones de patrones**: Caracteres alfabéticos, numéricos y alfanuméricos.
- **Reconocimiento de palabras reservadas**: Todas las palabras clave del lenguaje TDS25.
- **Reconocimiento de operadores**: Aritméticos, relacionales, lógicos y de asignación.
- **Reconocimiento de delimitadores**: Paréntesis, llaves, corchetes, punto y coma, comas.
- **Reconocimiento de literales**: Enteros (con validación de rango) y booleanos.
- **Reconocimiento de identificadores**: Siguiendo la convención de iniciar con letra o underscore.
- **Manejo de espacios y comentarios**: Descarte apropiado de elementos no significativos.

Analizador Sintáctico (sintaxis.y)
El analizador sintáctico fue implementado utilizando Bison y define:

- **Gramática completa**: Todas las producciones del lenguaje TDS25 según la especificación.
- **Tipos de datos**: Unión para manejar diferentes tipos de valores de tokens.
- **Precedencia y asociatividad**: Reglas para resolver ambigüedades en expresiones.
- **Manejo de errores**: Funciones personalizadas para reportar errores sintácticos.

1.5. Detalles de Implementación

Tokens y Expresiones Regulares

Los principales tokens reconocidos por el analizador léxico son:

Categoría	Token	Patrón/Valor
Palabras reservadas	PROGRAM	program
	INTEGER	integer
	INT	int
	BOOL	bool
	VOID	void
Operadores aritméticos	OP_SUMA	+
	OP_RESTA	-
	OP_MULT	*
	OP_DIV	/
	OP_RESTO	%
Operadores relacionales	OP_MAYOR	>
	OP_MENOR	<
	OP_COMP	==
Operadores lógicos	OP_AND	&&
	OP_OR	||
	OP_NOT	!
Delimitadores	PARA/PARC	( )
	LLAA/LLAC	{ }
	CORA/CORC	[ ]
Literales	INTEGER_LITERAL	[-]?{DIGIT}+
	TRUE/FALSE	true/false
Identificadores	ID	[a-zA-Z_][a-zA-Z0-9_]*

Gramática Implementada

La gramática implementada en Bison sigue fielmente la especificación de TDS25:

Reglas principales de la gramática

```
program
    : PROGRAM LLAA decl_list LLAC
    ;

decl_list
    : /* empty */
    | decl_list decl
    ;

decl
    : var_decl
    | method_decl
    ;

method_decl
    : TYPE ID PARA param_list_opt PARC block
    | VOID ID PARA param_list_opt PARC block
    | TYPE ID PARA param_list_opt PARC EXTERN PYC
    | VOID ID PARA param_list_opt PARC EXTERN PYC
    ;
```

Precedencia de Operadores
Se implementó la siguiente precedencia (de menor a mayor):

Definición de precedencia

```
%left OP_OR
%left OP_AND
%left OP_MAYOR OP_MENOR OP_COMP
%left OP_SUMA OP_RESTA
%left OP_MULT OP_DIV OP_RESTO
%right OP_NOT
%right UMINUS
```

1.6. Casos de Prueba

Se implementaron tres casos de prueba principales para validar el funcionamiento del compilador:

Example1.ctds: Prueba funciones con llamadas externas y flujo condicional con return

```
program {
    integer inc(integer x) {
        return x + 1;
    }

    integer get_int() extern;

    void print_int(integer i) extern;

    void main() {
        integer y = 0;
        y = get_int();
        if (y == 1) then {
            print_int(y);
        } else {
            return print_int(inc(y));
        }
    }
}
```

Example2.ctds: Validación de la ejecución de funciones externas

```
program {
    void main() {
        integer y = 0;
        y = get_int();
        if (y == 1) then {
            print_int(y);
        } else {
            print_int(inc(y));
        }
    }
}
```

Example3.ctds: Prueba bucles while y condiciones compuestas con operadores lógicos

```
program {
    void main() {
        integer x = 5;
        integer res = 0;
        while (x >= 0) {
            if (x != 0 && x >= 3) then {
                res = res+1;
            }
            x = x - 1;
        }
    }
}
```

ExampleError1.ctds: Valida detección de errores sintácticos

```
program {
    // Comprobacion de palabras reservadas juntas no validas
    void main() {
        integer y = 0;
        y = get_int();
        iftrue (y == 1) then {
            print_int(y);
        } else {
            return print_int(inc(y));
        }
    }
}
```

ExampleError2.ctds: Verifica manejo de errores

```
program {
    // Comprobacion de fuera de rango en tipo int/integer
    int x = 3000000000;
    return x;
}
```

ExampleError3.ctds: Comprueba detección de errores por comentarios no cerrados

```
program {
    /* Comprobacion de comentarios no cerrados */
    /* Como este comentario
}
```

1.7. Script de Compilación y Ejecución

Se desarrolló un script `script.sh` que automatiza el proceso completo de compilación y ejecución:

script.sh

```
#!/bin/bash
set -e

LEXER="lexico.l"
PARSER="sintaxis.y"
OUTPUT="program"

if [ -eq 0 ]; then echo "Uso: 0 Ejemplos/<archivo.ctds>"
    exit 1
fi

FILE=1 if [ ! -f "FILE"

]; then
    echo "Error: el archivo '
        FILE' no existe." exit 1 fi echo "==> Generando lexer con Flex..." flex "LEXER"









echo "==> Generando parser con Bison..."
bison -d -v "PARSER" echo "==> Compilando con GCC..." if [[ "OSTYPE"


== "darwin"* ]]; then
    gcc -o "OUTPUT" sintaxis.tab.c lex.yy.c seg cc -o "OUTPUT"

sintaxis.tab.c lex.yy.c -lfl
fi

echo "==> Ejecutando parser con FILE..." ./"OUTPUT"
< "
```

1.8. Resultados de Testing

Casos Exitosos

- **Example1.ctds**: Análisis completado sin errores
- **Example2.ctds**: Análisis completado sin errores
- **Example3.ctds**: Análisis completado sin errores

1.9. Próximas etapas del Proyecto

- **Analizador Semántico**
  Inicio: Lunes 15 de Septiembre.
  Entrega TS y AST: Miércoles 24 de Septiembre.
  Entrega An. Semántico: Miércoles 01 de Octubre.
- **Generador de Código Intermedio**
  Inicio: Miércoles 01 de Octubre.
  Entrega: Miércoles 08 de Octubre.
- **Generador de Código Objeto**
  Inicio: Miércoles 08 de Octubre.
  Entrega Enteros: Lunes 27 de Octubre.
- **Optimizador**
  Inicio: Lunes 27 de Octubre.
  Entrega: Miércoles 12 de Noviembre.
- **Entrega Final**
  Fecha: Viernes 15 de Noviembre.

1.10. Conclusiones

El **analizador léxico** reconoce correctamente todos los tokens especificados para el lenguaje, incluyendo palabras reservadas, operadores, delimitadores, literales e identificadores.
El **analizador sintáctico** verifica apropiadamente la estructura de los programas según la gramática especificada.

Los casos de prueba demuestran que el compilador maneja correctamente programas válidos, y el sistema de construcción automatizado facilita las pruebas y validación del código.

Esta base sólida nos permitirá seguir avanzando con confianza en la creación del **Compilador TDS25**.

En la siguiente etapa del proyecto trabajaremos en desarrollar el análisis semántico, en el cual se implementará la tabla de símbolos y las verificaciones de tipos, alcance y visibilidad de identificadores.